HST 952

Computing for Biomedical Scientists

Introduction

Medical informatics is interdisciplinary, and borrows concepts/ideas from:

- medicine
- computer science
- information science
- statistics

This course focuses mainly on computer science concepts used in medical informatics

Introduction

Why learn computer science basics?

- May need to design systems that others implement
- May need to implement prototypes
- May need to oversee programmers
- May need to analyze systems/products
- May need to apply these concepts in your research projects/future job

Course overview

Course Aims

- Present basic computer science concepts necessary for understanding and solving medical informatics problems (part 1)
- Introduce important data and knowledge representation methods (part 2)
- Examine data storage and retrieval issues as they relate to the medical domain (part 3)

Outline

- Overview of the computer
- Programming languages and paradigms
- Solving problems with a computer
  - abstraction
  - algorithms
- Questionnaires

Overview of the computer

1) Processor (CPU)

- **Datapath** (“brawn”: performs arithmetic operations)
- **Control** (“brain”: tells datapath, memory, I/O devices what to do according to the wishes of a program)

2) Memory

(where programs are kept when they are running; contains data needed by running programs)

Overview of the computer

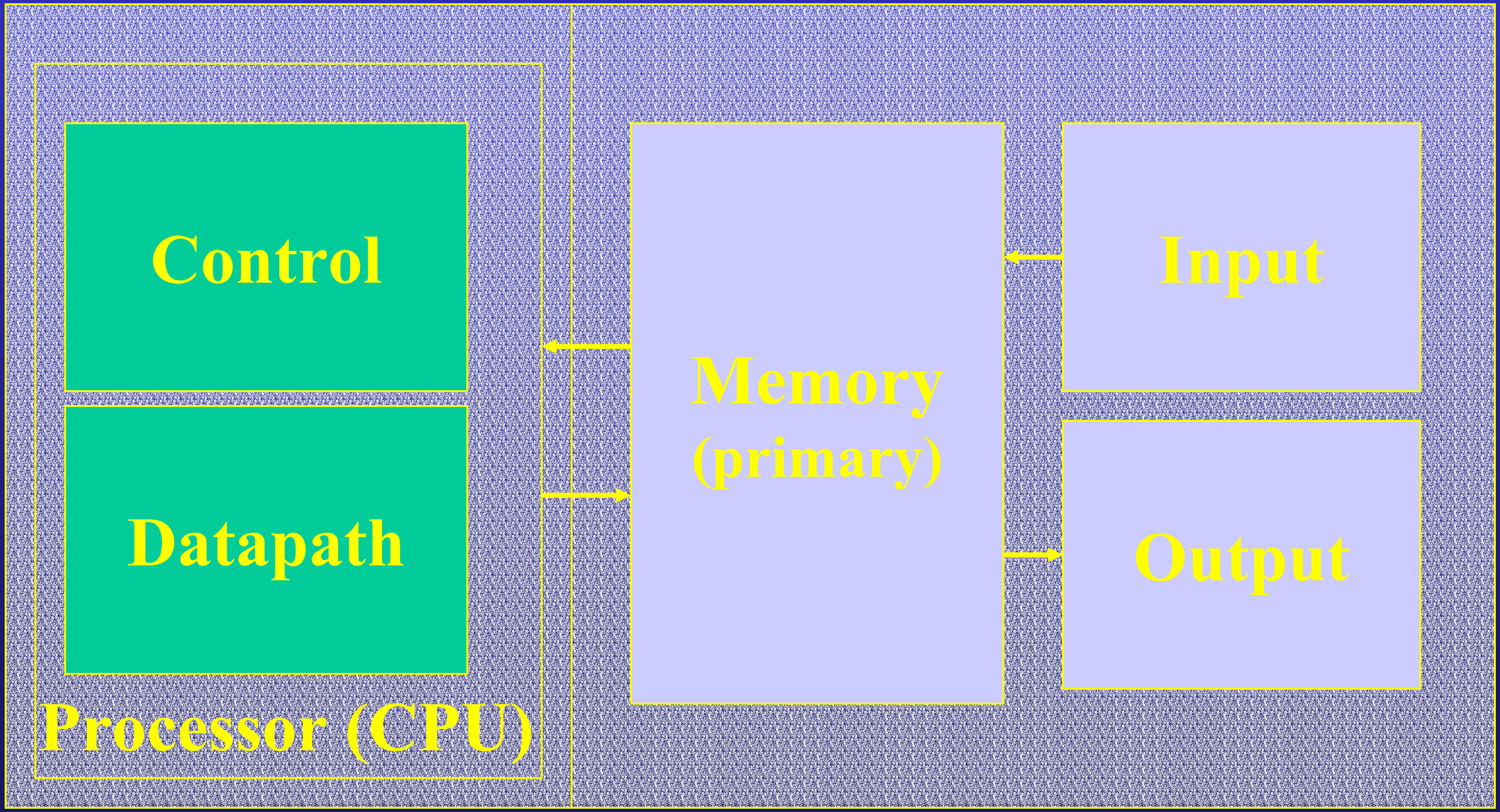
3) Input

(writes data to memory)

4) Output

(reads data from memory)

Overview of the computer



Overview of the computer

- Communication with a computer requires the use of electrical signals
- The easiest signals for a computer to understand are *off* and *on*
- The "alphabet" for a computer (used in its *machine language*) consists of two letters that represent the *off* and *on* signals: 0 and 1
- Each letter is referred to as a binary digit or **bit**

Overview of the computer

- Computers respond to/act on *instructions*
- Instructions are collections of bits that the computer understands
- Early programmers communicated with a computer in its machine language using binary numbers (very tedious)
- Communication using symbolic notation that was closer to the way humans think was soon adopted

Overview of the computer

- Programmers created a program called an *assembler* to translate from the new symbolic notation to binary
- This symbolic notation was called *assembly language*

The symbolic instruction: add A, B

is translated by the assembler to a form the computer understands: 1000110010100000

Overview of the computer

- Assembly language is still used today
- It requires a programmer to write one line for every instruction the computer will follow (think like a machine)
- Using the same ideas that led to assemblers, programmers created programs called *compilers*

Overview of the computer

- Compilers translate a high-level programming language (such as C++) to a low-level one (such as assembly language)

High-level statement:	A + B
Assembly language:	add A, B
Machine language:	1000110010100000

Overview of the computer

- The most important program that runs on any computer is the *operating system* (e.g., Unix/Linux, Mac OS, Windows XX, IBM OS/2)
- The OS manages the execution of every other program that runs on a computer
- It recognizes keyboard input, sends output to display screens, keeps track of files on disk and controls peripheral devices such as printers, etc.

Side note on Java

- Assembly language varies from one type of computer to another (as does machine language)
- Traditional compilers compile a program into the version of assembly language that runs on a particular type of computer
- Running the same program on another type of computer may require rewriting some of its high-level code and usually requires recompiling

Side note on Java

- Time is often spent creating versions of programs that are customized for a particular type of computer
- Java compilers translate high-level Java statements to *bytecode*
- Bytecode is not customized to any specific type of computer (it is *architecturally neutral*)
- The idea is that you can write a Java program on one type of computer and run it on any other

Programming language paradigms

Many high-level languages have the following characteristics:

- Programming in these languages is issuing instructions/commands (imperatives)
- There is a notion of modifiable storage (variables)
- Assignment is used to change the state of the variables (and consequently of the program)

Programming language paradigms

- Variables and assignment together serve as the programming language analog of hardware's modifiable storage (computer main memory)

This paradigm/pattern of programming is called *imperative programming*

Examples of imperative languages:

C, Pascal, C++, Java, Fortran

Programming language paradigms

Some other high-level languages have different characteristics:

- Programming in these languages is defining or declaring a solution
- A programming language analog of hardware's modifiable storage is not an important feature

This paradigm is called *declarative programming*

Examples of declarative languages:

Common Lisp, Scheme, Prolog

Example Problem

The factorial problem (for numbers greater than or equal to 0):

- the factorial of 0 is 1
- the factorial of any number, n, greater than 0 is n multiplied by the factorial of n minus 1

(finding the factorial of a number, n, larger than 0 requires finding the factorial of all numbers between n and 0)

Imperative Programming Example

Solution to the factorial problem for a number n:

- create a function called factorial-rec that takes n as an argument
- create a variable within factorial-rec called factorial
- create another variable within factorial-rec called temp
- if n is 0 **return a value of 1** (the factorial of 0 is 1)

otherwise (n is greater than 0)

- assign the value of n to factorial
- assign to temp the value of n minus 1
- return factorial multiplied by factorial-rec(temp)

Imperative Programming Example

Previous imperative algorithm in Java:

```
int factorial-rec(int n) // assume n is a number >= 0
{
   int factorial;
   int temp;
   if (n = 0) return 1; // factorial of 0 is 1
   factorial = n;
   temp = n - 1; // temp stores the next number in the series
   return  factorial * factorial-rec(temp);
}
```

Imperative Programming Example

Trace of factorial of the number 4:

factorial-rec(4) returns:

4 * factorial-rec(3) =

4 * 3 * factorial-rec(2) =

4 * 3 * 2 * factorial-rec(1) =

4 * 3 * 2 * 1 * factorial-rec(0) =

4 * 3 * 2 * 1 * 1 =

24

Declarative Programming Example

Declarative version of factorial in Scheme:

```
(define (factorial n)
        (if (= n 0) 1
        (* n (factorial (- n 1)))))
```

Trace of solution for the number 4:

```
(factorial 4) =
(* 4 (* 3 (* 2 (* 1 1)))) =
24
```

Observations

- Declarative programming is often a more intuitive approach to solving a problem
- Easier to use for certain classes of problems (theorem proving, etc.)
- However, since it does not attempt to mirror real hardware storage allocation, memory management and speed are sometimes problems

Solving problems with computers

In order to create a computer program to solve a particular problem we must:

- create a concise description/model of the problem, omitting details irrelevant to solving it (this is an example of *abstraction*)
- devise appropriate methods for solving this concise description (create an *algorithm*)

Solving problems with computers

- The word algorithm is named for al-Khowarizmi, a 9th century Persian mathematician
- An algorithm is a step by step procedure for solving a problem
- You are all familiar with clinical algorithms or algorithms for cooking (also known as recipes)
- Creating “elegant” algorithms (algorithms that are simple and/or require the fewest steps possible) is a principal challenge in programming

Creating a simple algorithm

Problem:

You need to share a pizza with five of your friends so that each of you gets a piece of the same size

Solution:

Volunteer?

Another algorithm

Problem:

Write out a step by step description of Euclid's algorithm for finding the greatest common divisor of two non-negative integers, X and Y:

As long as the value of neither X nor Y is zero, continue dividing the larger of the values by the smaller and assigning to X and Y the values of the divisor and remainder respectively. The final value of X when the remainder becomes 0 is the greatest common divisor.

Euclid’s GCD Algorithm

```
If x is less than or equal to 0 then stop
If y is less than or equal to 0 then stop

As long as y is greater than 0 repeat these steps:
    if x is greater than y then numerator = x and divisor = y
    otherwise numerator = y and divisor = x
    remainder  = numerator modulo divisor
    x  =  divisor
    y  = remainder
When y is equal to 0 the gcd is x
```

Solving problems with computers

- The set of steps that define an algorithm must be unambiguous (no room for misinterpretation)
- An algorithm must have a clear stopping point

  (a common mistake for programmers developing algorithms for problems that involve repetitive tasks is producing a series of steps that never end -- an infinite loop)

Another problem

A physician with a small practice wants you to write a program that calculates the number of minutes he can spend per appointment. He sees 15 patients each day and works an 8 hour day. He needs half an hour for lunch, and 10 minutes after each patient visit to write up notes. You may assume for this exercise that each patient gets an equal amount of time with the physician.

Questionnaires