CMPU 101 § 3

Problem-Solving and Abstraction

Fall 2020
"Honestly? I preferred when we didn’t talk about the elephant."

Shannon Wheeler, 2011
Course website

cs.vassar.edu/~cs101/3
Lectures:

Tuesday & Thursday, 1:30–2:45 p.m.
Sanders Physics 309 and Zoom

Lab:

Friday, 1:00 p.m.–3:00 p.m.
Zoom
Grading

10–12 lab exercises

15%
Grading

Approximately five programming homework assignments
Grading

- Two midterm exams: 25% each
- Labs: 15%
- Assignments: 20%
- Exam 1: 20%
- Exam 2: 20%
- Final: 20%

Two midterm exams
Grading

- Labs: 25%
- Assignments: 20%
- Exam 1: 20%
- Exam 2: 20%
- Final: 15%

*Regularly scheduled final exam*
HOW TO DESIGN PROGRAMS
Second Edition
An Introduction to Programming and Computing
Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Shriram Krishnamurthi
htdp.org
Hello, computer
We use computers every day as electronic black boxes that do amazing things by collecting, storing, retrieving, and transforming data.
Phones, tablets, desktops, watches, thermostats, medical devices, etc. are all computers.

Computers are devices that carry out the operations of a computation.
A *computation* is a series of numerical calculations and symbol manipulations.

**Numerical calculations include:**

- Basic arithmetic operations (add, subtract, multiply, divide)
- Basic trigonometric functions (sine, cosine, tangent)

**Symbolic manipulations include:**

- Logical comparison of numbers or symbols
- Decisions of what instructions to do next
- Substitutions of one string of letters and numbers for another
Amazing computations can be carried out when trillions of such simple operations are arranged in the proper order, e.g.,

- forecasting tomorrow’s weather
- deciding where to drill for oil
- designing the wings of an aircraft with enough lift to fly
- finding which physical places are most likely to be visited by a person
- calling for a taxi
- figuring out which two people would make a great couple 😘😍
This introductory course is titled *Problem-solving and Abstraction* because these are the essence of *computational thinking*. 
**Problem-solving**: Figuring out how to get computers to do a job for us.

This reflects the engineering tradition of computing, where people build methods and machines to help other people.

**Abstraction**: Explaining and interpreting the world as a complex of information processes.

This reflects the science tradition of computing in which people seek to understand how computation works and how it shows up in the world.
Algorithms and programs
Shopping, banking, travel, photography, movies, music are all powered by software running on our computers, in our pockets, in our houses, and in the cloud.
With the right programming, a computer is

- a movie theater
- a musical instrument
- a reference book
- a chess opponent
- ...

No other entity in the world has such an adaptable, universal nature, except a human.
The magic of a computer is its ability to become almost anything you can imagine...
The magic of a computer is its ability to become almost anything you can imagine...

...as long as you can explain exactly what that is.
The procedures that specify how the computer should do a job are called algorithms.
“An algorithm is a set of rules for getting a specific output from a specific input. Each step must be so precisely defined that it can be translated into computer language and executed by machine.”

Donald Knuth, 1977
Humans can carry out algorithms!

But we don’t do so nearly as fast as a machine can.

Modern computers can do a trillion steps in the time it takes a human to do one step.
For a computer to carry out an algorithm, it needs to be encoded as a *program* in a special-purpose language that’s translated into the low-level instructions that control a computer.
A *programming language* is a set of building blocks for constructing computer software.

Like a human language, a programming language has a vocabulary and a grammar.

How do you think a programming language is different?
When we program a computer to do something, everything needs to be described precisely.

Say you tell an accounting program to bill your clients the amount each owes.

Should the computer send a weekly bill for $0.00 to clients who owe nothing?

If you tell the computer to send a threatening letter to clients who haven’t paid, then clients who owe nothing will receive threatening letters until they send you a payment of $0!
Remember: Computers are incredibly dumb.

They perform mindless, mechanical steps extremely fast, but they have no understanding of what the steps mean.

The only errors they can correct are the ones you anticipate and provide with corrective algorithms.

You are the source of the intelligence. The computer amplifies your intelligence, but it has none of its own.
Programming languages

There are many more!

http://rigaux.org/language-study/diagram.html
There are many programming languages due to

intended use
history
habit
taste
Consider the turtle
Logo is simple enough to be used by a ten-year-old, but it embodies many of the features of the most sophisticated computer languages.

Including the ability to write programs that manipulate other programs!
think of a turtle holding a pen
forward 20, right 45, forward 20
Drawing pictures this way is fun but a lot of typing. Here’s where it gets interesting: *We can define new words.*
to `square`
  forward 20
  right 90
  forward 20
  right 90
  forward 20
  right 90
  forward 20
end

square
Once the word “square” has been defined, it becomes part of the computer’s vocabulary and can then be used to define other words, e.g.,

to window
  square
  square
  square
  square
  square
end

window
What if we want *big* squares and *little* squares?

The only thing that needs to change is the size, “10”

to **square** :size
  forward :size
  right 90
  forward :size
  right 90
  forward :size
  right 90
  forward :size
end
Other programs can call `square`; it doesn’t matter whether it’s a user-defined word or one of the language’s primitives.

In extending the language, the programmer uses the power of functional abstraction to create new building blocks.
What if we insert a word inside its own definition?

to design
  square
  right 10
  design
end
design
Cool, right?
Cool, right?

Not cool.

The recursive definition of “design” has a problem: it goes on forever.
A guru claimed the Earth was sitting on the back of a giant turtle.

“And what is the turtle sitting on?” asked a student.

“Another turtle”, replied the guru.

“And that turtle?” asked the student, beginning to grow skeptical.

“It’s no use asking”, said the guru. “It’s turtles all the way down.”
This particular infinite loop is easily avoided by writing the program with a parameter specifying how many squares to draw:

```plaintext
to design :number
  square
  right 10
  if :number > 0 [design :number - 1]
end
```
to tree :size
  forward :size
  if :size > 1 [two-trees :size / 2]
  back :size
end

to two-trees :size
  left 45
  tree :size
  right 90
  tree :size
  left 45
end
to tree :size
  forward :size
  if :size > 1 [two-trees :size / 2]
  back :size
end

to two-trees :size
  left 45
  tree :size
  right 90
  tree :size
  left 45
end
to tree :size
  forward :size
  if :size > 1 [two-trees :size / 2]
  back :size
end

to two-trees :size
  left 45
  tree :size
  right 90
  tree :size
  left 45
end

“A big tree is a stick with two smaller trees on top, but a little tree is just a stick.”
Recursion is powerful

Recursive definitions are convenient for specifying operations on recursive data.

Many of the types of data we like to manipulate – in particular, computer programs themselves – have recursive structures.

The typical recursive definition has two parts:

What’s to happen in a particular *simple case*

How a more *complex case* can be reduced to something simpler
Other programming languages differ from Logo in details of vocabulary and syntax, but they can all express the same kinds of procedures.

In the rest of this course, we’ll focus on a language called Racket. But you can imagine turtles.
All the way down
Computational thinking goes further than automation.

Information and computational processes have become a way of *understanding natural and social phenomena*.

Much of computational thinking today is oriented toward learning how the world works.
Many biologists, physicists, chemists, and other scientists look at their subject matter through a computational lens.

Professionals in the arts, humanities, and social sciences are also using computational thinking.

These are often called data science, digital humanities, or “computational X”, e.g., computational biology.
Computational thinking comprises ways of thinking and practicing that are sharpened and honed through practice.
How do you draw so well?

Practice.

It must be an innate gift... A gift from God...

It's practice.

I'll never understand how some people are so talented... A mystery...

Practice.

© Sarah Andersen
We’ve got a big journey ahead of us. I hope you’re excited!
Acknowledgments

This lecture incorporates material from:

Peter J. Denning and Matti Tedre
W. Daniel Hillis
Why aren’t we using Java? Python?

From the Preface of *How to Design Programs*:

Many people we encounter tell us they wish they knew how to code and then ask which programming language they should learn. Given the press that some programming languages get, this question is not surprising. But it is also wholly inappropriate. Learning to program in a currently fashionable programming language often sets up students for eventual failure. Fashion in this world is extremely short lived. A typical “quick programming in X” book or course fails to teach principles that transfer to the next fashion language. Worse, the language itself often distracts from the acquisition of transferable skills, at the level of both expressing solutions and dealing with programming mistakes.

In contrast, learning to design programs is primarily about the study of principles and the acquisition of transferable skills.