Harvey Mudd College

CS 60
Principles of Computer Science
Fall 2001
section 1

Bob Keller, Professor
keller@cs.hmc.edu
621-8483

IMPORTANT!
Special Orientation Sessions for Facilities

- The department staff have kindly put together orientation sessions to give you your accounts and acquaint you with our facilities:
  - Tonight (Wed.) at 8:30 p.m. in this room
  - Tomorrow night (Thurs.) at 8:30 p.m. in Gaileo/Pryne
- Attending *one* of these two sessions is **mandatory** if you don't have an account already.

Obligatory Course Reviews

- "Has everything: programming, hardware, theory, suspense, intrigue"
  - United CS Syndicate
- "A must-take course"
  - a CS major
- "Three thumbs up!"
  - Larry, Moe, and Curly

Instructor's Background

- Graduated from Hancock High School (St. Louis)
- B.S. in Engineering Science, M.S. in Electrical Engineering, Washington University (St. Louis)
- PhD in Electrical Engineering and Computer Sciences, U.C. Berkeley
- Taught at Princeton, University of Utah, U.C. Davis, Stanford
- V.P. R&D for a Software Company in Silicon Valley (1986-89)
- Department chair from 1991-2001
- Taught this course a few times before
- Wrote the book, got the T-shirt
- Design and develop software at JPL
- Also teaching Software Development (CS 121) and Jazz Improvisation (Music 84) this fall

---

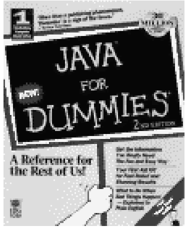
Office Hours (1249 Olin):

- Note: 1249 is in the southwest corner
- Tuesday, Thursday, 2-4, and others
- By drop-in (as available)
- Door usually closed, observe "IN" vs. "OUT"
- By appointment:
  - email keller@cs.hmc.edu
  - phone 621-8483
- Crisis center: 621-2373

Text

- *Computer Science: Abstraction to Implementation* (aka "*Computer Science for Smart People*") by Robert M. Keller
- Available from CS Department office, three options:
  - $30 for the 2001 edition (recommended)
  - $15 for the 1999 edition (while copies last)
  - web addition:
    http://www.cs.hmc.edu/~keller/cs60book
    Do not print major portions of book on HMC printers.

Boycott cheap imitations!



Auxiliary Text

- Some kind of Java reference, e.g. what you used in CS 5 or equivalent. Or check any bookstore for something that looks appealing.
- Won't need Java for a couple of weeks.

Help with Computer Account

- You will be given an account on turing.cs.hmc.edu.
- For problems with your account, you will need to contact either:
  - our system administrator,
    - Damon Rapp (drapp@cs.hmc.edu).
  - or one of our staff members:
    - staffnow@cs.hmc.edu
- I (Bob Keller) don't have the privileges necessary to set your password, etc.

How/Where to Login

- Room B102 is best (out the front lecture room door, up the stairs to the right, first door on the left)
- Remote is possible, **however**:
  - Must use secure ssh client, **not telnet**
  - For further information please see:
    http://www.cs.hmc.edu/tech_docs/qref/ssh.html
    This will tell you how to get free client for your machine.

Definition of Computer Science (CS)

Computer science involves synthesis and analysis of:

- Algorithms
- Information representations
- Communication processes
- Resource allocation methods
- Languages for all of the above

Role of CS

Computer science provides the *logical infrastructure* for the information-based society.

CS Characteristics and Contrasts

- CS not a “study of nature” as such
- We *create* what we study

  "The best way to predict the future is to invent it."
  Alan Kay
- Often abstract and mathematical:

  “Abstractions” *are* a product (e.g. API)
- Somewhat demanding in terms of precision and detail
- Self-applicable

Some **Mis**conceptions

- Computer Science is about studying computers

There is *some* of that, but CS is more generally about information and computation.

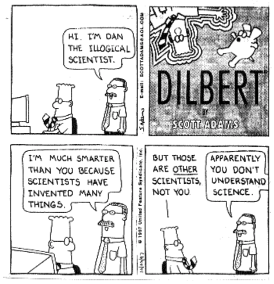
[Why isn’t surgery called “knife science”?]

Misconceptions (continued)

- Computer Science is just a “service” to other fields, not a real science.

Computer Science is an independent intellectual discipline that happens to enjoy applications to many other disciplines.

Dilbert meets the Scientist



Richard P. Feynman:

Computer science is not as old as physics; it lags by a couple of hundred years. However, this does not mean that there is significantly less on the computer scientist's plate that on the physicist's: younger it may be, but it has had a far more intense upbringing!

Misconceptions (continued)

- I could go on, but at the risk of rambling too much, I’ll stop here.

Broad Goals of CS 60

- Exposure to a variety of important areas of computer science
- Logical thinking and techniques
- Programming practice
- Specification and problem solving

CS 60 Goals
To learn important things about each of these areas:

- Data and information structuring
- Language syntax and parsing
- Programming paradigms
- Problem solving and specific algorithms
- Theoretical programming models
- User-interface building
- Various programming languages

(continued)

More CS 60 Goals

- Proposition logic
- Predicate logic
- Program complexity
- Finite-state machines
- Computer Architecture
- Exposure to parallel computing
- Limitations of computing

Why these things?

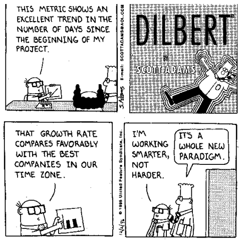
- They are basic to a range of fundamental areas.
- They are connected.
- They are interesting.
- They are accessible.

Goals wrt Programming

- Something about programming paradigms:
  - Functional programming
  - Object-oriented programming
  - Logic programming
  - Assembly-language programming
- because these are important in thinking about software and hardware construction.

What is a “paradigm” anyway?

- an example serving as a model
- a pattern



Alan Perlis:

A language that doesn't affect the way you think about programming is not worth knowing.

Why we write programs:

- to make a system or device that carries out some function
- to communicate with others
- to try ideas, to learn
- to *convince* ourselves we understand (rather than just saying we do)

Richard Hamming:

"The purpose of computing is insight, not numbers".

Alan Perlis:

"You *think* you know when you learn,

are *more sure* when you can write,

are *even more sure* when you can teach,

but are *certain* when you can program".

But if we haven't written the program ourselves, we haven't learned much.

Course Expectations

You can expect me to:

- produce written material and examples;
- create interesting and challenging problems;
- come to class prepared to discuss the problems, based on your questions;
- answer questions via email with a reasonable turn-around; and
- be available in my office for further questions.

I expect that you will:

- read the reading material;
- work on the assigned problems;
- start thinking about a programming assignment when it is first distributed, not the night before;
- come to class with questions and answers;
- retain a significant part of your knowledge through the final exam; and
- abide by the departmental honesty policy, as stated in

Getting Help

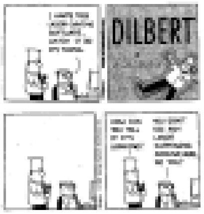
- I welcome you to come to my office for discussion of problems.
- The **grutors** will also be available for help.
- We (the grutors and I) want to receive your emailed questions.
- There is no stigma attached to getting help or asking questions. It is intended that you will need to do so.

Tips for Assignments

- **Start early** for best efficiency; then you can walk away if you get to a roadblock.
- Starting early will let your subconscious mind do some of the work.
- If you get to a roadblock, ask questions *before* spending hours trying to get around it.

Getting Help

- If a train station is where a train stops, what is a workstation?



Email Queries

- Send to:
  cs60help@cs.hmc.edu
- If it is relevant, we will broadcast our answers to your to the entire class.
- Your name will be *omitted* from the broadcast, unless you indicate that you prefer otherwise.
- It is **better** to email to cs60help than to a specific individual, to enhance the possibility of getting your question answered quickly.

Submitting Assignments

- You will be given an account on our UNIX server
  turing.cs.hmc.edu
- To submit homework, login to turing, and use the program:

  cs60submit *your-filename*

CS 60 Home Page

http://cs.hmc.edu/courses/current/cs60

- links to the syllabus
- will be updated constantly as we go

Overall Grade Breakdown

- 50% Assignments
- 25% Final Examination
- 10% Mid-term Examination
- 15% Class participation (worksheets, attendance, quizzes)

Point Breakdown in Assignments

- 50% for general correctness
- 25% for documentation (especially comments in code)
- 25% for goodness of approach, style, and robustness (subjective)

Late Assignment Policy

- You have an automatic 1-day grace period: you can submit the assignment up to midnight on the day following the due date.
- Example: Due date says January 29. You have until midnight on the night of January 30 to submit it correctly.
- Beyond this automatic grace period, late work is accepted only with a note from your Dean of Students.

Chapter One

- Gives an overview of the rest of the material
- Talks about abstraction

  It may take some time to appreciate this; maybe the whole semester, or more.

> "Truth be told, all software engineering is based on abstraction and abstract models.
>
> Abstract thinking is, developmentally speaking, a more advanced and sophisticated mode of thought that takes years for children to acquire. There are some adults who never learn to cut free of concrete literalism."

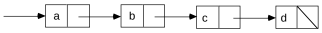
Larry Constantine
Object Magazine, Dec. 1996

Chapter Two

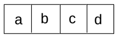
- Talks about information structures
- An abstract view of data structures
- Can be programmed directly in our rex language

Information Structures vs. Data Structures

- Information structures are an *abstraction* of data structures.
- Example: A “list”:
  - As a data structure, could be a linked list:



  - or it could be an array:



  - to give a few of the possibilities.
- I will go into linked lists on the board.

List Abstraction

- In an abstract sense, what matters most is the ***order*** of the elements in the list.
- We don’t have to say how the list is represented in the machine.
- We can just agree on some ***presentation*** or ***notation*** that shows this **order**, e.g.

[a, b, c, d]

Idea of “Structure”

- **Information** is composed of:
  - **Primitives**: *atomic* units of an agreed-upon universe, such as:
    - numbers
    - strings
  - **Structures**: *collections* of information, such as:
    - primitives
    - other structures

  possibly with additional ordering information

List Structures

- Lists with the each element of the same “type”:

[2, 3, 5, 7]

- The notation resembles one used for sets

{2, 3, 5, 7}

- except that:
  - Order matters with lists; it doesn’t for sets.
  - Duplication matters in lists; it doesn’t for sets.

Equality for Lists

- Two lists are defined to be ***equal*** when they have the same number of elements, and their elements occur in the same order.
- Examples:

[1, 2, 3] is equal to [1, 2, 3]

[1, 2, 3] is not equal to [3, 1, 2]

[1, 2, 3] is not equal to [1, 1, 2, 3]

The (one and only) Empty List

- The list with no elements
- The empty list is notated:

[ ]

Lists of Various Types of Elements

- List of integers:
  [-3, -2, -1, 0, 1, 2, 3]
- List of floats:
  [3.14, 6.0238e23, -0.4567]
- List of strings:
  [“Mary”, “had”, “a”, “little”, “dog”]

Mixing types of elements

- Can we mix types of elements?
  Yes!
- Should we mix types of elements?
  Maybe not if avoidable, but probably unavoidable in rex.

Specialized Uses of Lists

- Pairs:
  [1, 2]   [3, 4]   [5, 6]
- Triples:
  [1, 2, 3]   [4, 5, 6]
- n-tuples:
  [x_1, x_2, x_3, …, x_n]
  [y_1, y_2, y_3, …, y_n]

Implementing Sets with Lists

- A set is not a list, but
- a set can be *implemented* as a list:
  - simply ignore the ordering of the list, and
  - either:
    - ignore duplicates, or
    - guarantee no duplicates

called a representation invariant

- Ignoring duplicates has advantages, such as in element removal (why?)

Lists of Lists

- In order to keep track of, or manage, an arbitrary collection of lists, we can use lists with lists as elements
- List of pairs: [[1, 2], [3, 4], [5, 6]]
  - The ordering within each pair can be respected or not, as we desire (ordered vs. unordered pair)
- List of triples: [[1, 2, 3], [4, 5, 6]]
- List of assorted lists:
  [[1, 2, 3], [2, 3], [3], []]

Lists can be Nested Arbitrarily-Deeply

- List of lists of lists:
  [[[1, 2, 3], [2, 3]], [[3], []]]
- “Pyramidal” list:
  [[1, 2, 3, 4], [[1, 2], [3, 4]], [[[1, 2, 3, 4]]]]

Length of a List

- The length, or number of elements, in a list is the number at the "top level"

$$[\ \underbrace{[\,[1, 2, 3], [2, 3]\,]},\ \underbrace{[\,[3], []\,]}\ ]$$

has length 2

$$[\underbrace{[1, 2, 3, 4]},\ \underbrace{[[1, 2], [3, 4]]},\ \underbrace{[[[1, 2, 3, 4]]]}]$$

has length 3

Implementing
Other Information Structures
using Lists

Association Lists

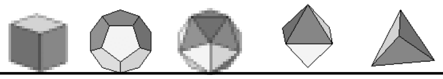
- An association list is a list of pairs.
  [["January", 31], ["February", 28], ["March", 31], ["April", 30]]
- Typically all first elements of the pairs are of the same type, as are all second elements.
- The pairs are not necessarily of the same type as each other.

Implementing an Ordered Dictionary

- A dictionary associates a value with each member of a set (called the domain).
- An ordered dictionary does this while keeping the domain ordered as well.
- A (finite) ordered dictionary can be implemented as an association list.

Ordered Dictionary Example

- Implement a dictionary of regular polyhedra as an association list:
  - With each name is associated a pair:
    [number-of-faces, number-of-sides-per-face]
- [["cube", [6, 4]],
  ["dodecahedron", [12, 5]],
  ["icosahedron", [20, 3]],
  ["octahedron", [8, 3]],
  ["tetrahedron", [4, 3]]



Binary Relations

- A binary relation is a set of ordered-pairs, with the elements drawn from a common set.
- A finite set can be implemented as a list.
- An ordered-pair can be represented as a list.
- Therefore, a finite binary relation can be represented as a list.

Example Binary Relation Implementation

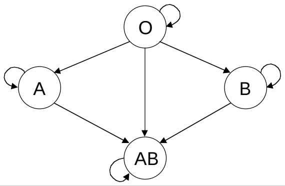
- Consider the binary relation “can be donor for” on the set of blood types: {A, B, AB, O}
- As a list, this could be represented
  [[“A”, “A”], [“A”, “AB”], [“B”, “B”], [“B”, “AB”], [“AB”, “AB”], [“O”, “A”], [“O”, “AB”], [“O”, “B”], [“O”, “O”]]

Directed Graphs

- A Directed Graph may be viewed as a way of presenting a binary relation:
  - The nodes of a directed graph correspond to the elements in the domain
  - The arcs (arrows) of a directed graph correspond to the pairs that are related

Directed Graph Example

- For the binary relation can be donor for represented as a list previously
  [[“A”, “A”], [“A”, “AB”], [“B”, “B”], [“B”, “AB”], [“AB”, “AB”], [“O”, “A”], [“O”, “AB”], [“O”, “B”], [“O”, “O”]]
  the directed graph would be

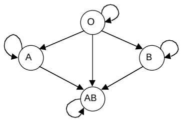


Other Representations

- As you will explore in the text and homework, this is not the only way to represent binary relations.
- It is not the best way for all, or even most, applications.

Properties of Binary Relations (1 of 2)

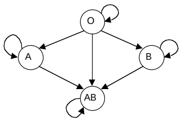
- The previous relation example illustrates two common properties that a binary relation may have:
  - transitive property: For every x, y, z * in the domain
    if x is related to y and y is related to z,
    then x is related to z.
  - reflexive property: For every x in the domain
    x is related to x.



* (Note that any of x, y, z may be equal.)

Properties of Binary Relations (2 of 2)

- The previous example has only the second of the following additional two common properties:
  - symmetric property: For every x, y in the domain
    if x is related to y then y is related to x.
  - anti-symmetric property: For every x, y in the domain
    if x is related to y and y is related to x, then
    x = y.



Inferred Properties of Binary Relations?

- Which of the following are true for binary relations?
  - If the relation has the transitive and symmetric property, then it also has the reflexive property.
  - A relation cannot have both the symmetric and anti-symmetric property.

Additional Terminology

- A relation with the reflexive, symmetric, and transitive properties is called an equivalence relation. Such a relation generalizes the notion of equality, since in this case if x is related to z and y is related to z, then x is related to y.

  In other words, if each of a set of elements is related to a common thing, the elements in the set and the common thing are all related to each other.
- A relation with the reflexive, anti-symmetric, and transitive properties is called a partial order. (See if you can see why.)

Example of an Equivalence Relation

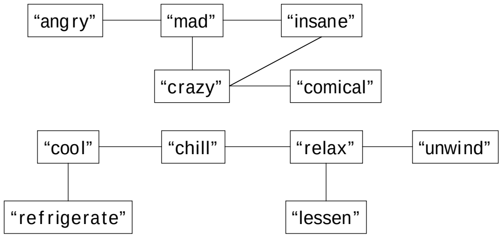
- Consider the relation “sounds the same as” on a set of words, such as {“air”, “ere”, “heir”, “buy”, “by”, “bye”, “dew”, “do”, “due”, “ewe”, “you”, “yew”}
  - Reflexive: Every x is a homophone of x.
  - Symmetric: If x is a homophone of y then y is a homophone of x.
  - Transitive: If x is a homophone of y and y is a homophone of z, then x is a homophone of z.
- Therefore this is an equivalence relation.

Undirected Graphs

- An undirected graph is a way of presenting a symmetric binary relation: Since whenever x is related to y also y is related to x, we don't have to show direction with arcs. Instead of calling them arcs then, it is common to call them edges.

Undirected Graph Example

- An example of a symmetric relation and its undirected graph is “is a synonym of”:



More Information Structures?

- There is a **lot** more to be said, and our next topic will be trees.
- But for now, we will discuss some ways to work with these representations in an actual **language**.

Functional Programming

- Functional programming is one of the major **fundamental** programming paradigms.
- It means programming only by composing functions, not using assignment statements.
- It can be used in conjunction with other paradigms, such as object-oriented programming.

Functional Programming is “Complete”

- There is a certain well-defined sense in which a programming language can be called “complete”:
  - The language is capable of representing *any* computable function.
  - Most languages of significance, including most functional ones, are complete in this sense.
- More on the definition of “computable” and “complete” later.

A Functional Programming Language

- We will use the language rex to exemplify functional programming.
- rex is interactive:
  - definitions are entered
  - expressions are evaluated to get results
- You may run rex on turing:
  - `unix > rex`
  - `rex >`

UNIX prompt

rex prompt

---

rex usage examples
(user input is shown in bold)

```
rex > length([ [1, 2], [3, 4], [5, 6] ]);
3

rex > sort([3, 9, 1, 2, 8, 7, 5, 6, 4]);
[1, 2, 3, 4, 5, 6, 7, 8, 9]

rex > sort(["oats", "peas", "beans", "barley"]);
[barley, beans, oats, peas]

rex >
```

---

more rex usage examples
(define variables to avoid re-entry)

```
rex > x = [3, 9, 1, 2, 8, 7, 5, 6, 4];
1

rex > x;
[3, 9, 1, 2, 8, 7, 5, 6, 4]

rex > sort(x);
[1, 2, 3, 4, 5, 6, 7, 8, 9]

rex > x;
[3, 9, 1, 2, 8, 7, 5, 6, 4]
```

This 1 means true, the definition was accepted.

---

more rex usage examples
(previous session continued)

```
rex > length(x);
9

rex > reverse(x);
[4, 6, 5, 7, 8, 2, 1, 9, 3]

rex > append(x, x);
[3, 9, 1, 2, 8, 7, 5, 6, 4,
 3, 9, 1, 2, 8, 7, 5, 6, 4]

rex >
```

Load files to prevent re-typing

contents of file text.rex, prepared with a text editor, such as Emacs:

```
// This is a set of rex definitions, with comments

// x is a list of some small random numbers.

x = [3, 9, 1, 2, 8, 7, 5, 6, 4];


// y is a list of some grains.

y = sort(["oats", "peas", "beans", "barley"]);


// z is a list of pairs

z = [ [1, 2], [3, 4], [5, 6] ];


/*
   Above you see comments to end-of-line.
   You can also have multi-line comments such as this one,
       just like Java or C++.
*/
```

At least two ways to load a file:

Method 1: Include the file name on the UNIX command line:

here

```
unix > rex test.rex
test.rex loaded
rex > x;
[3, 9, 1, 2, 8, 7, 5, 6, 4]

rex > y;
[barley, beans, oats, peas]

rex > z;
[[1, 2], [3, 4], [5, 6]]
```

You can re-run the command without retyping, e.g. `unix > !r`

At least two ways to load a file:

Method 2: Include the file from a rex command line

here

```
unix > rex
rex > *i test.rex
read file test.rex
rex > x;
[3, 9, 1, 2, 8, 7, 5, 6, 4]

rex > y;
[barley, beans, oats, peas]

rex > z;
[[1, 2], [3, 4], [5, 6]]
```

Split-screen editing in Emacs

(what I use most of the time)

UNIX shell in emacs window

In Emacs:
`control-x 2` to split window
`escape-x shell` to get shell
Can cut/paste using only keystrokes

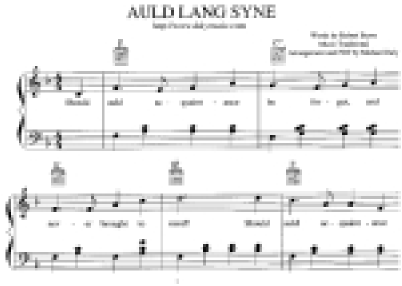
your rex file for editing

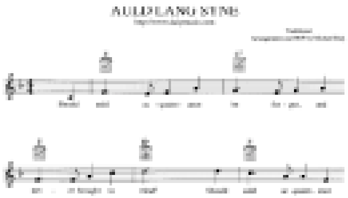
Abstraction Exercise

- For discussion next time:
  - Think up and describe an area outside of CS where you (or others) use abstraction.

Abstraction Exercise

- Example: Music
  - Full score
  - Lead sheet
  - Chords only
  - Chord functions, or
  - Keys only

Full Score

Lead Sheet



Chords Only

Auld Lang Syne

F C7 F Bb F

Chord Functions

Auld Lang Syne

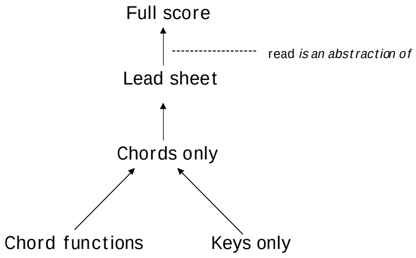
I V^7 I IV I

Keys Only

Auld Lang Syne

F / / Bb F

Abstraction Graph


Full score

read *is an abstraction of*

Lead sheet

Chords only

Chord functions

Keys only