Harvey Mudd College

CS 60
Principles of Computer Science
Spring 2002
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.) from 7:00 to 7:15 in this room
  - Tomorrow night (Thur.) from 7:00 to 7:15
- Attending one of these sessions is **mandatory** if you don't have an account already.

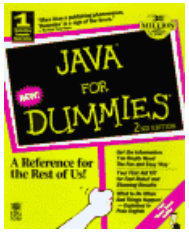
My Office Hours (1249 Olin):

- Note: 1249 is in the southwest corner of Olin
- 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.
- Reading Assignment:
  - Chapters 1 and 2

*Boycott demeaning books!



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, quota, 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.

Strong Recommendation

- Learn your way around UNIX as quickly as possible.
- Learn to use Emacs (text editor) as quickly as possible.
  - Emacs is a powerful, life-long tool.

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 (That's us, folks!).

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
- "Abstractions" *are* often a product
  (e.g. Java awt: Abstract Window Toolkit)

Some Misconceptions

- Computer Science is about studying computers

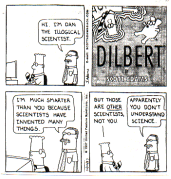
  There is *some* of that, but CS is more generally about information and computation.

  Analog: Surgery is "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.



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!

Broad Goals of CS 60

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

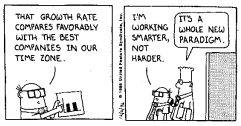
Goals wrt Programming

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

  Alan Perlis

What is a "paradigm" anyway?

- what it used to take to make a call from a phone booth :)
- an example serving as a model
- a pattern



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)

Old Chinese Proverb

- I hear, I forget.
- I see, I remember.
- I program, I understand.

My Best Advice

- CS 60 assignments are not cook-book; they demand a certain level of intellectual engagement. Therefore:
- Starting thinking about an assignment as soon as it is given.
- Let your sub-conscious mental processes help solve problems.
- Allow yourself space to experiment.

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: Mail to: cs60help@cs.hmc.edu
- Get help as soon as possible when you feel you are not making progress; don't waste hours not knowing what to do.
- There is no stigma attached to getting help or asking questions. It is intended that you will need to do so.

Policy on Stupid Questions

- The only "stupid questions" are the ones that don't get asked.

Submitting Assignments

- You will be given an account on our UNIX server

  turing.cs.hmc.edu
- You can *only* submit homework from this account using the program:

  cs60submit *your-filename*

CS 60 Home Page

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

- links to the syllabus and other good stuff
- will be updated constantly as we go

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."

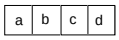
Larry Constantine
Object Magazine, Dec. 1996

Chapter Two

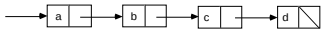
- Talks about information structures
- An abstract view of data structures
- Can be programmed directly in our *rex* language

Information Structures vs. Data Structures

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



  - or could be a linked list:



  - to give a few of many possibilities.
- Each of these is an *implementation* of the abstraction.

List Abstraction

- In an abstract sense, what matters most in a list is the ***order*** of the elements.
- 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

    regarded as "indivisible" for the current discussion.
  - **Structures**: *collections* of information, such as:
    - primitives
    - other structures

    possibly with imposed ordering information

List Structures

- Lists with the each element of the same "type":
  [2, 3, 5, 7]
- The notation resembles ones you've seen 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?
  Not if avoidable, but sometimes unavoidable in rex.

Specialized Uses of Lists

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

Implementing Set Abstraction using 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 vs. guaranteeing have advantages and disadvantages (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"
  [[[1, 2, 3], [2, 3]], [[3], []]]
  has length 2

  [[1, 2, 3, 4], [[1, 2], [3, 4]], [[[1, 2, 3, 4]]]]
  has length 3

Implementing Other Information Structures using Lists

Association Lists

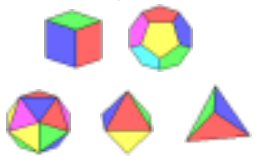
- An association list is a list of pairs.
  [["January", 31], ["February", 28], ["March", 31], ["April", 30]]
- Typically all first elements of the pairs are of the same type, as are all second elements.
- The pairs are not necessarily of the same type as each other.

Implementing an Ordered Dictionary

- A dictionary is an abstraction associating 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]]
  ]



Using a Dictionary rex function *assoc*

- The built-in function assoc behaves as follows:
  - It has two arguments:
    - The first argument is a member of a domain, say D.
    - The second argument is an association list with domain D.
  - The result is the first pair in the association list in which the first element matches the first argument.
  - If there is no match, [ ] is returned.
    ([ ] is not a pair, so the meaning is clear.)

Example using assoc:

```
// Definition of polyhedra

polyhedra =
    [ ["cube",         [6, 4]],
      ["dodecahedron", [12, 5]],
      ["icosahedron",  [20, 3]],
      ["octahedron",   [8, 3]],
      ["tetrahedron",  [4, 3]]
   ];

// Expression to be evaluated

assoc("octahedron", polyhedra);

// Expected result:

["octahedron", [8, 3]]
```

Using the rex builtin 2-ary test function

```
//   Expression to be tested          Desired result
test(assoc("octahedron", polyhedra), ["octahedron",   [8, 3]]);
```

Sample session:

```
turing ~:1> rex polyhedra.rex
ok: assoc("octahedron", [[cube, [6, 4]], [dodecahedron, [12, 5]],
 [icosahedron, [20, 3]], [octahedron, [8, 3]],
 [tetrahedron, [4, 3]]]) ==> [octahedron, [8, 3]]
polyhedra.rex loaded
1 rex > ^D                  Control-D means end-of-input
turing ~:1>
```

If there were an error, it would be indicated with "bad" instead of "ok" and the error count would be reported at the end.

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 of lists of 2 elements each.

Example Binary Relation Implementation

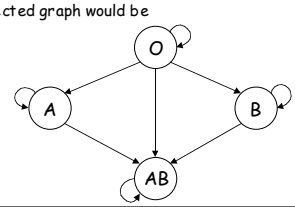
- Consider the binary relation "can be donor for" on the set of blood types: {A, B, AB, O}
- As a list, this could be represented
  [["A", "A"], ["A", "AB"],
  ["B", "B"], ["B", "AB"],
  ["AB", "AB"],
  ["O", "A"], ["O", "AB"], ["O", "B"], ["O", "O"]]
- Is this meaningful as an association list?

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

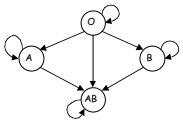


Other Representations

- 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.

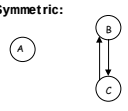


*** (Note that any of x, y, z may be equal!)**

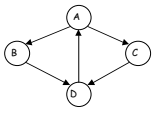
Properties of Binary Relations (2 of 2)

- 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.

Symmetric:

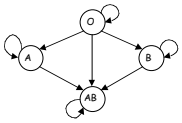


Anti-symmetric:



Partial Orders

- A relation with the reflexive, anti-symmetric, and transitive properties is called a partial order.
- Example:



Inferred Properties of Binary Relations?

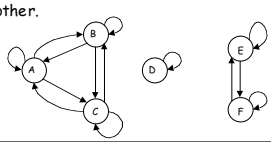
- Which of the following are true for binary relations?
  - If the relation has the transitive and symmetric property, then it also has the reflexive property.
  - A relation cannot have both the symmetric and anti-symmetric property.

Equivalence Relations

- 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.

Equivalence:



Example of an Equivalence Relation

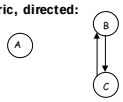
- Consider the relation "is a ***homophone*** of" ("sounds the same as") on a set of words, such as {"air", "ere", "heir", "buy", "by", "bye", "dew", "do", "due", "ewe", "you", "yew"}
  - Reflexive: Every x is a homophone of x.
  - Symmetric: If x is a homophone of y then y is a homophone of x.
  - Transitive: 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

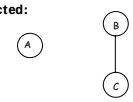
- An undirected graph is a way of presenting a symmetric binary relation:

  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.

Symmetric, directed:

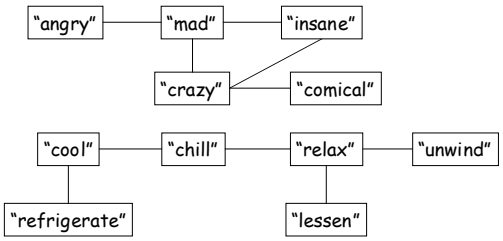


undirected:



Undirected Graph Example

- An example of a symmetric relation and its undirected graph is "can be a synonym of":



Abstraction Exercise

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

More Information Structures?

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

Functional Programming

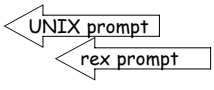
- Functional programming is one of the major **fundamental** programming paradigms.
- It means programming only by composing functions, not using assignment statements.
- It can be used in conjunction with other paradigms, such as object-oriented programming.

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:
  - `turing > rex`
  - `rex >`



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
```

This 1 means true, the definition was accepted.

```
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]
```

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]
```

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:

```
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]]
```

here

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

```
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]]
```

here

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