General Data Characterizations

- Abstraction: the data from a behavioral viewpoint (what can be done with the data)
- Representation: the data as represented in the computer (how the behaviors are implemented)
- Presentation: the data as presented to the user (what we see)

Example: Natural Numbers

- Presentation: Decimal numerals:
  
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...

- Representation: Binary:
  
  0000, 0001, 0010, 0011, 0100, 0101, 0110, ...

- Abstraction: Peano axioms

Peano Axioms (1889)

- N designates the set of natural numbers
- 0 is a particular natural number in N
- S is a function S: N → N, such that:
  - For all x ∈ N, S(x) ≠ 0.
  - For all x, y ∈ N, S(x) = S(y) implies x = y.
- If P is any predicate, such that
  - P(0)
  - For all x ∈ N, P(x) implies P(S(x))
  - then for all x ∈ N, P(x).

Peano vs. Decimal Presentation

- 0 is decimal is Peano 0
- 1 is S(0)
- 2 is S(S(0))
- Number n in general is S(S(...S(0)))... n S's
- 0 ≠ 1, S(1) ≠ S(2), etc.
To fill the space of a square with a single curve:
Start with a line across the diagonal.
Define operation X:
Replace a line segment with a curve segment as shown:
Iterate X ad infinitum

As a result of this process, the line segments eventually fill the entire square.

Aside: Peano's Space-Filling Curve

Peano’s Space-Filling Curve

Peano’s Space-Filling Curve

Peano’s Space-Filling Curve

Peano’s Space-Filling Curve

Peano’s Space-Filling Curve

Peano’s Space-Filling Curve

The Purpose of Abstraction in CS

- To abstract means to eliminate irrelevant detail.
- This is vital in presenting simple, crisp, specifications of what software does.
- It is also useful in hiding detail from those who don’t need to know about it:
  - They can’t mess with it.
  - One can change the details without changing the concept.

Abstract Art: Similar meaning, but not the same purpose

Two paintings of a tree by Mondrian.
Abstractions in Disciplines
- Chemistry is an abstraction of Physics.
- Biology is an abstraction of Chemistry.
- Genetics is an abstraction of Biology.
- Why did these abstractions evolve?

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

Open-List Abstraction
- An extension of Peano ideas
- Provides a way to create and manipulate lists in a programming language
- All definitions have a mathematical basis.
- In the text, we refer to
  - information structures (abstraction and presentation) vs.
  - data structures (implementation and representation).

Information Structures vs. Data Structures
- Information structures are an abstraction of data structures.
- Example: A "list" information structure, to give a few of many possible data structures:
  - could be an array
    - \[ a \ b \ c \ d \]
  - or could be a linked list
    - \[ a \rightarrow b \rightarrow c \rightarrow d \]
  - Each of these is a representation or implementation of the abstraction.

Array vs. List Implementation
- Array advantages:
  - Constant time access to any element based on the index of the element
  - Less storage space, since don't store pointers
- Linked list advantages:
  - Does not require contiguous memory locations to hold array elements; uses fragmented memory more efficiently.
  - Less expensive to insert or remove items.

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, possibly with imposed ordering information.

List Structures

- Lists notation (presentation) we will often use:
  \([2, 3, 5, 7]\)
- The notation resembles ones you’ve seen for sets
  \([2, 3, 5, 7]\)
- Distinctions:
  - 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: \([\ ]\)
- Also called the “null list”

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:
  \(\["Mary", \"had", \"a", \"little", \"dog\]\)
Specialized Uses of Lists

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

Implementing Set Abstraction using Lists

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

Lists can be Nested Arbitrarily-Deeply

- List of lists of lists:
  \[[[[[1, 2, 3, [2, 3]], [3], [3]]]]\]
- Lists of lists during "sort by repeated merging":
  \[[[3], [8], [5], [1], [2], [7], [6], [4]], [3, 8], [1, 5], [2, 7], [4, 6]], [1, 3, 5, 8], [2, 4, 6, 7]], [1, 2, 3, 4, 5, 6, 7, 8]]

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-size lists:
  \[[[1, 2, 3], [2, 3], [3], []]\]

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

The member function

- member tells whether a specified element is in a specified list. It returns 1 or 0 accordingly:

  - member(11, [5, 7, 11, 13]) \(\equiv\) 1
  - member(12, [5, 7, 11, 13]) \(\equiv\) 0
  - member(3, [[[1, 2, 3], [2, 3]], [3], []]] \(\equiv\) 0
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, and all second elements are of the same type.
- 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:
  ```
  ["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:

```javascript
// 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 ~> 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 ~> !

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