6.001 SICP
Data abstraction revisited

- Data structures: association list, vector, hash table
- Table abstract data type
- No implementation of an ADT is necessarily "best"
- Abstract data types are a technique for information hiding
  - in the types as well as in the code

CONCRETE

ABSTRACT

CONCEPT

Table: a set of bindings
- binding: a pairing of a key and a value
- Abstract interface to a table:
  - make: create a new table
  - put! key value: insert a new binding
  - get key: look up the key, return the corresponding value
- This definition IS the table abstract data type
- Code shown later is a particular implementation of the ADT

Examples of using tables

<table>
<thead>
<tr>
<th>People</th>
<th>Age</th>
<th>Job</th>
<th>Pay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fred</td>
<td>34</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>John</td>
<td>48</td>
<td>48</td>
<td>48</td>
</tr>
<tr>
<td>Bill</td>
<td>2000</td>
<td>1999</td>
<td>1998</td>
</tr>
</tbody>
</table>

Values associated with keys might be data structures
Values might be shared by multiple structures

Traditional LISP structure: association list
- A list where each element is a list of the key and value.
- Represent the table

\[
\text{People: } \left( \begin{array}{c|c}
\text{key} & \text{value} \\
\hline
x & 15 \\
\hline
y & 20 \\
\end{array} \right)
\]

as the alist:

\[
\left( (x, 15) (y, 20) \right)
\]

Alist operation: find-assoc

\[
\begin{align*}
&\text{(define (find-assoc key alist)} \\
&\hspace{1cm} (\text{cond}) \\
&\hspace{2cm} ((null? alist) #f) \\
&\hspace{2cm} ((\text{equal?} \ \text{key} \ \text{(caar} \ \text{alist})) \ \text{(cadar} \ \text{alist})) \\
&\hspace{2cm} (\text{else (find-assoc key (cdr} \ \text{alist}))))
\end{align*}
\]

\[
\begin{align*}
&\text{(define a1 '((x 15) (y 20)))} \\
&\text{(find-assoc 'y a1) \Rightarrow 20}
\end{align*}
\]

An aside on testing equality

- = tests equality of numbers
- Eq? Tests equality of symbols
- As will see, also tests equality of list structures
- Equal? Tests equality of symbols, numbers or lists of symbols and/or numbers that print the same
- Eqv? Tests equality of list as actual structures, not just prints the same
Alist operation: add-assoc

(define (add-assoc key val alist)
  (cons (list key val) alist))

(define a2 (add-assoc 'y 10 a1))

a2  ==>  ((y 10) (x 15) (y 20))

(find-assoc 'y a2)  ==>  10

We say that the new binding for y “shadows” the previous one – you can see how the find-assoc procedure does this.
Example 2: Pair/List Mutation

```
(define x (list 'a 'b))
```

- How mutate to achieve the result at right?

```
(set-car! (cdr x) (list 1 2))
```

1. Eval `(cdr x)` to get a pair object
2. Change car pointer of that pair object

How do we know Table1 is an ADT implementation

- Potential reasons:
  - Because it has a type tag No
  - Because it has a constructor No
  - Because it has mutators and accessors No

- Actual reason:
  - Because the rest of the program does not apply any functions to Table1 objects other than the functions specified in the Table ADT
  - For example, no `car`, `cdr`, `map`, `filter` done to tables

- The implementation (as an Alist) is hidden from the rest of the program, so it can be changed easily

Information hiding in types: opaque names

- Opaque: type name that is defined but unspecified
- Given functions `m1` and `m2` and unspecified type `MyType`:
  - `(define (m1 number) ...) ; number → MyType`
  - `(define (m2 myt) ...) ; MyType → undef`
- Which of the following is OK? Which is a type mismatch?
  - `(m2 (m1 10))`
  - `(car (m1 10))`
- Effect of an opaque name:
  - no functions will match except the functions of the ADT

Types for table1

- Here is everything the rest of the program knows

```
Table1<k,v> opaque type 
make-table1 void → Table1<anytype,anytype>
table1-put! Table1<k,v>, k, v → undef
table1-get Table1<k,v>, k → (v | null)
```

- Here is the hidden part, only the implementation knows it:

```
Table1<k,v> = symbol x Alist<k,v>
Alist<k,v> = list<k x v x null>
```

Lessons so far

- Association list structure can represent the table ADT
- The data abstraction technique (constructors, accessors, etc) exists to support information hiding
- Information hiding is necessary for modularity
- Modularity is essential for software engineering
- Opaque type names denote information hiding
Hash tables

- Suppose a program is written using Table1
- Suppose we measure that a lot of time is spent in table1-get
- Want to replace the implementation with a faster one
- Standard data structure for fast table lookup: hash table
- Idea:
  - keep N association lists instead of 1
  - choose which list to search using a hash function
    - given the key, hash function computes a number x where 0 ≤ x ≤ (N-1)

Example hash function

- A table where the keys are points
  - point
  - graphic object
  - (5,5) (circle 4)
  - (10,6) (square 8)

(define (hash-a-point point N)
  (modulo (+ (x-coor point) (y-coor point)) N))

; modulo x n = the remainder of x + n
; 0 ≤ (modulo x n) ≤ n-1 for any x

Hash function output chooses a bucket

Store buckets using the vector ADT

- Vector: fixed size collection with indexed access
  - vector<A> opaque type
  - make-vector number, A → vector<A> speed access
  - vector-ref vector<A>, number → A
  - vector-set! vector<A>, number, A → undef
  - (make-vector size value) ===> a vector with size locations; each initially contains value
  - (vector-ref v index) ===> whatever is stored at that index of v
    (error if index >= size of v)
  - (vector-set! v index val) stores val at that index of v
    (error if index >= size of v)

Table2: Table ADT implemented as hash table

(define t2-tag 'table2)
(define (make-table2 size hashfunc)
  (let ((buckets (make-vector size nil)))
    (list t2-tag size hashfunc buckets)))
(define (size-of tbl) (cadr tbl))
(define (hashfunc-of tbl) (caddr tbl))
(define (buckets-of tbl) (cadddr tbl))

- For each function defined on this slide, is it
  - a constructor of the data abstraction?
  - an accessor of the data abstraction?
  - an operation of the data abstraction?
  - none of the above?

get in table2

(define (table2-get tbl key)
  (let ((index ((hashfunc-of tbl) key (size-of tbl))))
    (find-assoc key
      (vector-ref (buckets-of tbl) index))))

- Same type as table1-get
put! in table2

```
(define (table2-put! tbl key val)
  (let ((index
        ((hashfunc-of tbl) key (size-of tbl)))
        (buckets (buckets-of tbl)))
        (vector-set! buckets index
          (add-assoc key val
                    (vector-ref buckets index)))))
```

- Same type as table1-put!

Table2 example

```
(define tt2 (make-table2 4 hash-a-point))
/table2-put! tt2 (make-point 5 5) 20
/table2-put! tt2 (make-point 5 7) 15
/table2-get tt2 (make-point 5 5)
```

Is Table1 or Table2 better?

- Answer: it depends!
  - Table1: make put! extremely fast
get O(n) where n=number of calls to put!
  - Table2: make put! space N where N=specified size
get must compute hash function compute hash function plus O(n)
where n=average length of a bucket

- Table1 better if almost no gets or if table is small
- Table2 challenges: predicting size, choosing a hash function that spreads keys evenly to the buckets

Summary

- Introduced three useful data structures
  - association lists
  - vectors
  - hash tables
- Operations not listed in the ADT specification are internal
- The goal of the ADT methodology is to hide information
- Information hiding is denoted by opaque type names

```
(define (add-assoc key val alist)
  (cons (list key val) alist))
(define (add-assoc key val alist)
  (cons (list key val) alist))
(define table1-tag 'table1)
(define (make-table1) (cons table1-tag nil))
(define (table1-get tbl key)
  (find-assoc key (cdr tbl)))
(define (table1-put! tbl key val)
  (set-cdr! tbl (add-assoc key val (cdr tbl))))
(define (make-table2 size hashfunc)
  (let ((buckets (make-vector size nil)))
    (list t2-tag size hashfunc buckets)))
(define (table2-get tbl key)
  (let ((index
          ((hashfunc-of tbl) key (size-of tbl))))
    (find-assoc key
               (vector-ref (buckets-of tbl) index))))
(define (table2-put! tbl key val)
  (let ((index
          ((hashfunc-of tbl) key (size-of tbl))))
    (vector-set! buckets index
                 (add-assoc key val
                            (vector-ref buckets index))))
```