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

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 replaces any previous binding of that key
  - 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

People
Fred
John
Bill
Age
2000
1999
1998
Job
Pay

Traditional LISP structure: association list
- A list where each element is a list of the key and value.
- Represent the table as the alist:
  `((x 15) (y 20))`

Alist operation: find-assoc
(define (find-assoc key alist)
  (cond
    ((null? alist) #f)
    ((equal? key (caar alist)) (cadar alist))
    (else (find-assoc key (cdr alist)))))

(define a1 '((x 15) (y 20)))
(find-assoc 'y a1)  ==> 20

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

Alists are not an abstract data type

• Missing a constructor:
  • Use quote or list to construct
    (define al `((x 15) (y 20)))

• There is no abstraction barrier:
  • Definition in scheme language manual:
    "An alist is a list of pairs, each of which is called an
    association. The car of an association is called the key."

• Therefore, the implementation is exposed. User may operate
  on alists using list operations.

/filter (lambda (a) (< (cadr a) 16)) a1))
===> ((x 15))

Why do we care that Alists are not an ADT?

• Modularity is essential for software engineering
  • Build a program by sticking modules together
  • Can change one module without affecting the rest

• Alists have poor modularity
  • Programs may use list ops like filter and map on alists
  • These ops will fail if the implementation of alists change
  • Must change whole program if you want a different table

• To achieve modularity, hide information
  • Hide the fact that the table is implemented as a list
  • Do not allow rest of program to use list operations
  • ADT techniques exist in order to do this

Table1: Table ADT implemented as an 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))))

Compound Data

• constructor:
  (cons x y) creates a new pair p

• selectors:
  (car p) returns car part of pair
  (cdr p) returns cdr part of pair

• mutators:
  (set-car! p new-x) changes car pointer in pair
  (set-cdr! p new-y) changes cdr pointer in pair
  ; Pair,anytype -> undef -- side-effect only!

Example 1: Pair/List Mutation

(define a (list 1 2))
(define b a)
(a) (1 2)
(b) (1 2)
(set-car! a 10)
b ==> (10 2)

Compare with:
(define a (list 1 2))
(define b (list 1 2))
(set-car! a 10)
b ==> (1 2)
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)) ; return type of m1 matches argument type of m2
  (car (m1 10)) ; return type of m1 fails to match argument type of car

- 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 × 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

<table>
<thead>
<tr>
<th>point</th>
<th>graphic object</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5,5)</td>
<td>(circle 4)</td>
</tr>
<tr>
<td>(10,6)</td>
<td>(square 8)</td>
</tr>
</tbody>
</table>

(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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vector&lt;A&gt;</code></td>
<td>opaque type</td>
</tr>
<tr>
<td><code>make-vector</code></td>
<td>number, A → vector&lt;A&gt; access</td>
</tr>
<tr>
<td><code>vector-ref</code></td>
<td>vector&lt;A&gt;, number → A</td>
</tr>
<tr>
<td><code>vector-set!</code></td>
<td>vector&lt;A&gt;, number, A → undef</td>
</tr>
</tbody>
</table>

- (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**

```scheme
(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**

```scheme
(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 extremely fast
    - put! extremely fast
    - get O(n) where n=# calls to put!
  - Table2: make space N where N=specified size
    - put! must compute hash function
    - get 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