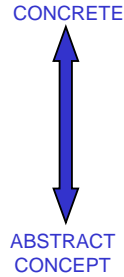


6.001 SICP Data abstraction revisited

- 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



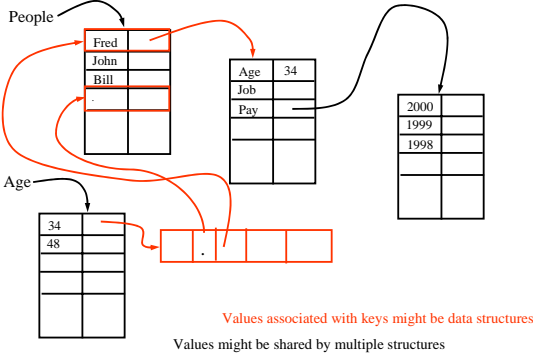
1

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

2

Examples of using tables



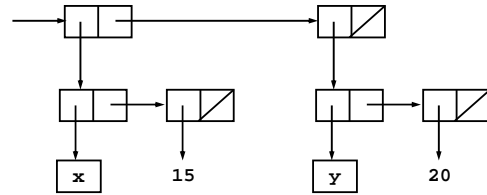
3

Traditional LISP structure: association list

- A list where each element is a list of the key and value.
- Represent the table

| |
|---------------|
| x : 15 |
| y : 20 |

as the alist: ((x 15) (y 20))

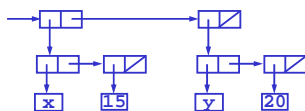


4

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 al '((x 15) (y 20)))
(find-assoc 'y al) ==> 20
```



5

An aside on testing equality

- =** tests equality of numbers
- Eq?** Tests equality of symbols
- Equal?** 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

6

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

7

Alists are not an abstract data type

- Missing a constructor:
 - Use `quote` or `list` to construct

```
(define a1 '((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))
```

8

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

9

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))))
```

10

Compound Data

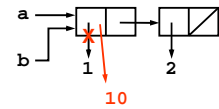
- **constructor:**
`(cons x y)` creates a new pair `p`
- **selectors:**
`(car p)` returns car part of pair
`(cdr p)` returns cdr part of pair
- **mutators:**
`(set-car! p new-x)` changes car pointer in pair
`(set-cdr! p new-y)` changes cdr pointer in pair
; Pair, anytype -> undef -- side-effect only!

11

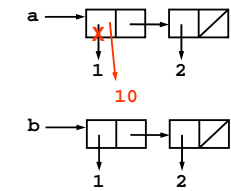
Example 1: Pair/List Mutation

```
(define a (list 1 2))
(define b a)
a → (1 2)
b → (1 2)

(set-car! a 10)
b ==> (10 2)
```



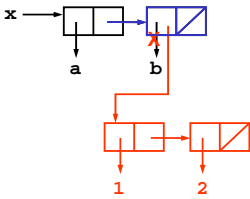
```
Compare with:
(define a (list 1 2))
(define b (list 1 2))
(set-car! a 10)
b → (1 2)
```



12

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

13

Table1 example

```
(define (table1-get tbl key)
  (find-assoc key (cdr tbl)))

(define tt1 (make-table1))
(table1-put! tt1 'y 20)
(table1-put! tt1 'x 15)
(table1-get tt1 'y)
```

```
(define (make-table1)
  (set-cdr! tbl
            (add-assoc key val (cdr tbl))))

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

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

14

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

15

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
               ; car: pair<A,B> → A
```
- Effect of an opaque name:
 - no functions will match except the functions of the ADT

16

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 × v × null >
```

17

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

18

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 $0 \leq x \leq (N-1)$

19

Example hash function

- A table where the keys are points

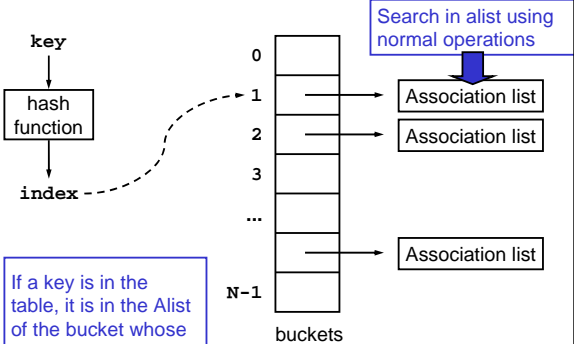
| point | graphic object |
|--------|----------------|
| (5,5) | (circle 4) |
| (10,6) | (square 8) |

```
(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

20

Hash function output chooses a bucket



21

Store buckets using the vector ADT

- Vector: fixed size collection with indexed access

vector<A> opaque type Vector has constant speed access
make-vector number, A → vector<A>
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)

22

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) (caddrtbl 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?

23

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

24

