

For example, (let ((a 10) (b 20)) $(+ a b)) \Rightarrow 30$ Using a **let**, the definition of **reval1**, a version of **rev** that reverses all levels of a list, is easy: (define (revall L) (if (null? L) ь (let ((E (if (list? (car L)) (revall (car L)) (car L)))) (append (revall (cdr L)) (list E))))) (revall '((1 2) $(3 4))) \Rightarrow$ ((4 3) (2 1))

CS 538 Spring 2008

125

Subsets

Another good example of Scheme's recursive style of programming is subset computation.

Given a list of distinct atoms, we want to compute a list of all subsets of the list values.

For example,

 $\begin{array}{ccc} (\text{subsets} & (1 \ 2 \ 3)) \implies \\ (\ () \ (1) \ (2) \ (3) \ (1 \ 2) \ (1 \ 3) \\ (2 \ 3) \ (1 \ 2 \ 3)) \end{array}$

The order of atoms and sublists is unimportant, but all possible subsets of the list values must be included.

Given Scheme's recursive style of programming, we need a recursive definition of subsets.

That is, if we have a list of all subsets of n atoms, how do we extend this list to one containing all subsets of n+1 values?

First, we note that the number of subsets of n+1 values is exactly *twice* the number of subsets of n values.

For example,

(subsets '(1 2)) \Rightarrow (() (1) (2) (1 2)), which contains 4 subsets.

(subsets '(1 2 3)) Contains 8 subsets (as we saw earlier).

Moreover, the extended list (of subsets for n+1 values) is simply the list of subsets for n values *plus* the result of "distributing" the new value into each of the original subsets.

126

```
Thus (subsets (1 2 3)) \Rightarrow
(()(1)(2)(3)(12)(13)
  (2 \ 3) \ (1 \ 2 \ 3)) =
( )
      (1)
            (2) (1 2) ) plus
((3)(13)(23)(123))
This insight leads to a concise
program for subsets.
We will let (distrib L E) be a
function that "distributes" E into
each list in L.
For example,
(distrib '(() (1) (2) (1 2)) 3) =
((3)(31)(32)(312))
(define (distrib L E)
  (if (null? L)
      ()
      (cons (cons E (car L))
            (distrib (cdr L) E))
   )
)
```

CS 538 Spring 2008

We will let (extend L E) extend a list **L** by distributing element **E** through **L** and then appending this result to **L**. For example, (extend '(() (a)) 'b) \Rightarrow (()(a)(b)(b a)) (define (extend L E) (append L (distrib L E)) Now **subsets** is easy: (define (subsets L) (if (null? L) (list ()) (extend (subsets (cdr L)) (car L))))

CS 538 Spring 2008

128

129

DATA STRUCTURES IN SCHEME

In Scheme, lists and S-expressions are basic. Arrays can be simulated using lists, but access to elements "deep" in the list can be slow (since a list is a linked structure).

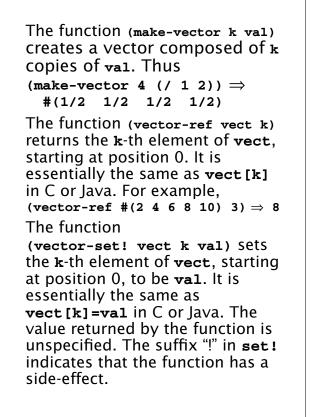
To access an element deep within a list we can use:

- (list-tail L k) This returns list L after removing the first k elements. For example, (list-tail '(1 2 3 4 5) 2) ⇒ (3 4 5)
- (list-ref L k) This returns the k-th element in L (counting from 0). For example, (list-ref '(1 2 3 4 5) 2) \Rightarrow 3

Vectors in Scheme

Scheme provides a vector type that directly implements one dimensional arrays. Literals are of the form #(...) For example, #(1 2 3) or #(1 2.0 "three") The function (vector? val) tests whether val is a vector or not. (vector? 'abc) \Rightarrow #f (vector? '(a b c)) \Rightarrow #f (vector? #(a b c)) \Rightarrow #t The function (vector v1 v2 ...) evaluates v1, v2, ... and puts them into a vector.

(vector 1 2 3) \Rightarrow #(1 2 3)



```
CS 538 Spring 2008
```

132

For example, (define v #(1 2 3 4 5)) (vector-set! v 2 0) $\mathbf{v} \Rightarrow \#(1\ 2\ 0\ 4\ 5)$ Vectors *aren't* lists (and lists aren't vectors). Thus (car #(1 2 3)) doesn't work. There are conversion routines: • (vector->list V) converts vector **v** to a list containing the same values as v. For example, (vector->list $\#(1\ 2\ 3)) \Rightarrow$ (1 2 3)• (list->vector L) converts list L to a vector containing the same values as **L**. For example, $(list \rightarrow vector '(1 2 3)) \Rightarrow$ #(1 2 3)

CS 538 Spring 2008

 In general Scheme names a conversion function from type T to type Q as T->Q. For example, string->list converts a string into a list containing the characters in the string.

Records and Structs

In Scheme we can represent a record, struct, or class object as an *association list* of the form ((obj1 val1) (obj2 val2) ...) In the association list, which is a list of (object value) sublists, object serves as a "key" to locate the desired sublist.

For example, the association list

((A 10) (B 20) (C 30)) serves the same role as

struct

{ int a = 10; int b = 20; int c = 30;}

134

133

```
The predefined Scheme function
(assoc obj alist)
checks alist (an association list)
to see if it contains a sublist with
obj as its head. If it does, the list
starting with obj is returned;
otherwise #f (indicating failure) is
returned.
For example,
(define L
 '( (a 10) (b 20) (c 30) ) )
(assoc 'a L) ⇒ (a 10)
(assoc 'b L) ⇒ (b 20)
(assoc 'x L) ⇒ #f
```

```
We can use non-atomic objects as
    kevs too!
    (define price-list
       '( ((bmw m5)
                           71095)
          ((bmw z4))
                           40495)
           ((jag xj8)
                           56975)
          ((mb s1500)
                           86655)
        )
    )
    (assoc '(bmw z4) price-list)
         \Rightarrow ((bmw z4) 40495)
CS 538 Spring 2008
```

CS 538 Spring 2008

```
Using assoc, we can easily define
a structure function:
(structure key alist) Will
return the value associated with
key in alist; in C or Java
notation. it returns alist.key.
(define
  (structure key alist)
  (if (assoc key alist)
    (car (cdr (assoc key alist)))
    #f
  )
)
We can improve this function in
two ways:
• The same call to assoc is made
 twice; we can save the value
 computed by using a let
 expression.
• Often combinations of car and cdr
 are needed to extract a value.
```

Scheme has a number of predefined functions that combine several calls to **car** and **cdr** into one function. For example,

 $(caar x) \equiv (car (car x))$ $(cadr x) \equiv (car (cdr x))$ $(cdar x) \equiv (cdr (car x))$ $(cddr x) \equiv (cdr (cdr x))$

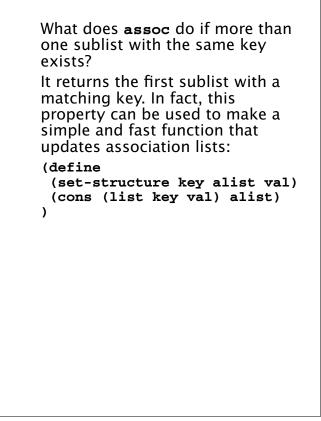
Using these two insights we can now define a better version of **structure**

```
(define
 (structure key alist)
 (let ((p (assoc key alist)))
  (if p
       (cadr p)
       #f
    )
)
)
```

138

136

137



If we want to be more spaceefficient, we can create a version that updates the internal structure of an association list, using **set-cdr!** which changes the **cdr** value of a list: (define (set-structure! key alist val) (let ((p (assoc key alist))) (if p (begin (set-cdr! p (list val)) alist) (cons (list key val) alist))))

CS 538 Spring 2008

140

CS 538 Spring 2008