Set data abstractions

Represent a set as a list of its elements in which no element appears more than once.

\[
\begin{align*}
\text{define} & \quad (\text{element-of-set?} \ x \ \text{set}) \\
& \quad (\text{cond} \ ((\text{null?} \ \text{set}) \ \text{false}) \\
& \quad \quad \quad ((= \ x \ (\text{car} \ \text{set})) \ \text{true}) \\
& \quad \quad \quad ((\text{element-of-set?} \ x \ (\text{cdr} \ \text{set}))) \ )
\end{align*}
\]

\[
\begin{align*}
\text{define} & \quad (\text{element-of-set?} \ x \ \text{set}) \\
& \quad (\text{not} \ (\text{null?}) \\
& \quad \quad \quad (\text{filter} \ \text{lambda} \ (e) \ (\text{equal?} \ e \ x) \ \text{set}))
\end{align*}
\]

Adjoin-set

If the object to be adjoined is already in the set, we just return the set. Otherwise, cons it on!

\[
\begin{align*}
\text{define} & \quad (\text{adjoin-set} \ x \ \text{set}) \\
& \quad (\text{if} \ (\text{element-of-set?} \ x \ \text{set}) \\
& \quad \quad \quad \text{set} \\
& \quad \quad \quad (\text{cons} \ x \ \text{set}))
\end{align*}
\]

We can define union-set in one line using adjoin-set:

\[
\begin{align*}
\text{define} & \quad (\text{union-set} \ \text{set1} \ \text{set2}) \\
& \quad (\text{accumulate} \ \text{adjoin-set} \ \text{set1} \ \text{set2})
\end{align*}
\]

Intersection-set

Assume we know how to form the intersection of set2 and the cdr of set1...

\[
\begin{align*}
\text{define} & \quad (\text{intersection-set} \ \text{set1} \ \text{set2}) \\
& \quad (\text{cond} \ ((\text{or} \ (\text{null?} \ \text{set1}) \ (\text{null?} \ \text{set2})) \ \text{'}()) \\
& \quad \quad \quad ((\text{element-of-set?} \ (\text{car} \ \text{set1}) \ \text{set2}) \\
& \quad \quad \quad \quad (\text{cons} \ (\text{car} \ \text{set1}) \\
& \quad \quad \quad \quad \quad (\text{intersection-set} \ (\text{cdr} \ \text{set1}) \ \text{set2}))) \\
& \quad \quad \quad (\text{else} \ (\text{intersection-set} \ (\text{cdr} \ \text{set1}) \ \text{set2})))
\end{align*}
\]

Alternatively, use filter:

\[
\begin{align*}
\text{define} & \quad (\text{intersection-set} \ \text{set1} \ \text{set2}) \\
& \quad (\text{filter} \ \text{lambda} \ (x) \ (\text{element-in-set?} \ x \ \text{set2}) \ \text{set1}))
\end{align*}
\]

Complexity

Element-of-set? may have to scan the entire set.

Intersection-set does an element-of-set? check for each element of set1

The complexity of intersection/union is:

\[O(n^2)\] for two sets of size \(n\)
Ordered intersection-set

But this really helps with intersection-set.

Unordered representation this operation required O(n^2) steps, because we performed a complete scan of set2 for each element of set1.

But with the ordered representation, we can use a linear method....how?

```
(define (intersection-set set1 set2)
  (if (or (null? set1) (null? set2))
    '()
    (let ((x1 (car set1)) (x2 (car set2)))
      (cond ((= x1 x2)
            (cons x1
               (intersection-set (cdr set1)
                                (cdr set2))))
            ((< x1 x2)
             (intersection-set (cdr set1) set2))
            ((< x2 x1)
             (intersection-set set1 (cdr set2)))))))
```

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))`
Alist operation: find-assoc

\[
\text{(define \( \text{find-assoc} \) \( \text{key} \) \( \text{alist} \))}
\]
\[
\text{(cond}
\begin{align*}
\text{((null? \( \text{alist} \)) #f)} \\
\text{((equal? \text{key} \text{(caar \( \text{alist} \))}) \text{(cadar \( \text{alist} \))})} \\
\text{(else \( \text{(find-assoc} \text{key} \text{(cdr \( \text{alist} \)))})})
\end{align*}
\]
\[
\text{(define \( a1 \) \('((x 15) (y 20))')}\text{)}
\]
\[
\text{(find-assoc 'y a1) ==» 20}
\]

Alist operation: add-assoc

\[
\text{(define \( \text{add-assoc} \) \( \text{key} \) \( \text{val} \) \( \text{alist} \))}
\]
\[
\text{(cons \( \text{list} \text{key} \text{val} \) \( \text{alist} \))}
\]
\[
\text{(define \( a2 \) \( \text{add-assoc} \text{'y} \text{10} a1 \))}
\]
\[
a2 ==» (\text{('(y 10) (x 15) (y 20))})
\]
\[
\text{(find-assoc 'y a2) ==» 10}
\]

find-assoc-entry

\[
\text{(define \( \text{find-assoc-entry} \) \( \text{key} \) \( \text{alist} \))}
\]
\[
\text{(cond}
\begin{align*}
\text{((null? \( \text{alist} \)) #f)} \\
\text{((equal? \text{key} \text{(caar \( \text{alist} \))}) \text{(car \( \text{alist} \))})} \\
\text{(else \( \text{(find-assoc-entry} \text{key} \text{(cdr \( \text{alist} \)))})})
\end{align*}
\]
\[
\text{(find-assoc-entry 'bar '((foo 6) (bar #f) (ed 4)) ==» (bar #f))}
\]

Alists are not an abstract data type

- Missing a constructor:
  - Use quote or list to construct
    \[
    \text{(define \( a1 \) \('((x 15) (y 20))')}\text{)}
    \]
- 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.
    \[
    \text{(filter (lambda (a) (\text{<} (\text{cadar a}) 16)) \( a1 \)) ==» (\text{('x 15))}}\text{)}
    \]
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

---

**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 replacing 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 an implementation of the ADT

---

**Table 1: Table ADT implemented as an Alist**

```scheme
(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))))
```

**Examples of using tables**

---

**Table 2: Table ADT implemented as hash table**

```scheme
(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))
```

---

**ADT and tags**

- standard pattern for an ADT with tagged data
  1. a variable in the ADT implementation stores the tag
  2. attach the tag in the constructor
  3. write a predicate that checks the tag
  4. operations strip the tags, operate, attach the tag again
- Use tagged data everywhere (including return values)
- Using tagged data is only defensive programming
  - if you check the tags and don't use the else branch
- Traditionally, ADT operations / accessors don't check tags, but paranoia is fine here.
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)))))

(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)))))

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))
(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))

Hash functions
What makes a good hash function?
- not too many indicies
- indicies are evenly distributed
- hash function is easy to compute

E.g., how to hash products at Wal-mart?
- price?
- price/100?
- price modulo 100?
- SKU?