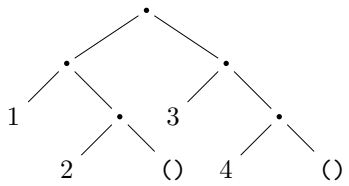
# CS 450

# Lecture 4: The `quote` Special Form; More Data Structures

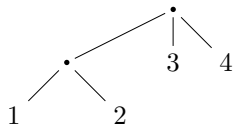
Carl D. Offner

## 1 Trees

`cons` cells (i.e., pairs) can be used to construct binary trees, or in fact more general trees. For instance, the list `((1 2) 3 4)` is represented internally as




but you can think of it as




In either case, we would write this as the Scheme expression

```
==> (list (list 1 2) 3 4)
```

That is, we have here three different representations of the same thing:

- The data structure as you (the user) think of it.
- The way you represent this data structure as a Scheme expression.
- The way that Scheme represents this expression internally.

The way a Scheme expression is translated to the Scheme internal form is fixed. But the way you use Scheme expressions to represent your own data structures is up to you. The only thing you have

to make sure of is that you can translate consistently back and forth between the way you think of your data structure and the Scheme expression for it (and the Scheme representation of it).

Suppose we want to count the leaves in such a tree. For our purposes here, the empty list terminal nodes don't count as leaves, but all the other terminal nodes do. We can write a Scheme function to count the leaves as follows:

```
(define (countleaves x)
  (cond ((null? x) 0)
        ((not (pair? x)) 1)
        (else (+ (countleaves (car x))
                 (countleaves (cdr x)) )) ))
```

Now suppose we make the following definitions:

```
(define x (list 1 2 3))
(define y (list 4 5 6))
```

Watch what happens:

```
==> (append x y)

(1 2 3 4 5 6)
==> (cons x y)

((1 2 3) 4 5 6)
==> (list x y)

((1 2 3) (4 5 6))
```

## 2 quote

This is a special form that inhibits evaluation. It can be abbreviated as follows:

```
(quote <exp>)   <=>  '<exp>
```

Thus, the following are all equivalent:

```
(quote (1 2 3 4))  ==>  (1 2 3 4)
'(1 2 3 4)         ==>  (1 2 3 4)
(list 1 2 3 4)     ==>  (1 2 3 4)
```

On the other hand, suppose we define the following symbols:

```
(define a 1)
(define b 2)
(define c 3)
(define d 4)
```

Then we have

```
(quote (a b c d))   ==>  (a b c d)
'(a b c d)          ==>  (a b c d)
(list a b c d)      ==>  (1 2 3 4)
```

(`list` is a procedure; it evaluates its arguments.)

With the same definitions of `a` and `b`, here are some more examples:

```
==> (list a b)

(1 2)
==> (list 'a 'b)

(a b)
==> (list 'a b)

(a 2)
==> '(a b)

(a b)
==> (car '(a b c))

a
==> (cdr '(a b c))

(b c)
```

# 3 Equality

There are three tests for equality in Scheme. Corresponding to these, there are three tests to see whether an object is a member of a list, and there are also three selectors to find elements in a lookup table:

```
   eq?        eqv?       equal?
   memq       memv       member
   assq       assv       assoc
```

`member` is defined like this:

```
(define (member item x)
  (cond ((null? x) #f)
        ((equal? item (car x)) x)
        (else (member item (cdr x))) ))
```

Thus, `member` evaluates to the sublist of `x` starting with `item`, if there is one, and to the empty list otherwise. `memq` and `memv` are defined similarly. Note that the names of these three procedures don't end with "?"—this is because they do not generally return Booleans.

The semantics of the three forms of equality tests are as follows:

`eq?` checks only for pointer equality. That is, `(eq? <exp1> <exp2>)` checks to see whether `<exp1>` and `<exp2>` evaluate to objects that are at the same location in memory. This is often not true, and the result may surprise you. For example, `(eq? 2 2)` evaluates to `#f` on most systems.

`eqv?` is the same, except that numbers, characters, and symbols are now guaranteed to be "correct". That is, `(eqv? 2 2)` is guaranteed to be `#t`.

`equal?` checks to see that its two arguments after evaluation are structurally the same, and that the leaves are `eqv?` (In particular, a string is regarded as built up out of characters, and so strings are handled "correctly".)

In practice, you should use `equal?` and `member`. The other forms are pretty much of historical interest only, and often lead to confusion and bugs. So just don't use them. See the discussion in R$^5$RS for more information, if you are interested in this.

## 3.1 The all-purpose lookup procedure: `assoc`

An *association list* is a list of pairs. It is used as a lookup table, where the car of each pair is the lookup key and the cdr is the value being looked up. For instance, suppose we had the following association list:

```
(define alist
  (list '(skyscraper . noun) '(go . verb) '(from . preposition) '(road . noun)))
```

The idea is that this represents a table holding the information that `from` is a `preposition`, that `go` is a `verb`, and so on.. The function `assoc` is used to lookup elements in the table. You have to be careful though, because it doesn't return the value—it returns the whole pair. So for instance

```
(assoc 'from alist)
```

evaluates to the pair `(from . preposition)`. So in this case, the value you are looking for is really the `cdr` of what is returned. `assoc` is a primitive procedure in Scheme, but we could easily write it like this:

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

There are also versions `assq` and `assv` that use `eq?` and `eqv?` instead of `equal?`, but you will never need to use them.

The `assoc` function is very simple, but it is used all over the place in Scheme. It's very important.

It's actually very common—and our text does this almost all the time—to construct these lookup tables not as pairs but as 2-element lists, like this:

```
(define alist
  (list '(skyscraper noun) '(go verb) '(from preposition) '(road noun)))
```

With this kind of list `(cdr (assoc 'from alist))` would evaluate to the single-element list `(preposition)`, so you would have to do something like this to look up a value:

```
(let ((value-as-list (assoc key alist)))
  (cadr value-as-list))
```

# 4 Representing sets

We need procedures that include the following:

- `union`
- `intersection`
- `element-of?`
- `adjoin`

## 4.1 Representing sets as unordered lists

```
(define (element-of? x set)
  (cond ((null? set) #f)
        ((equal? x (car set)) #t)
        (else (element-of? x (cdr set))) ))
```

`element-of?` is $O(n)$.

```
(define (adjoin x set)
  (if (element-of? x set)
      set
      (cons x set) ))
```

`adjoin` is $O(n)$.

```
(define (intersection set1 set2)
  (cond ((or (null? set1) (null? set2)) '())
        ((element-of? (car set1) set2)
         (cons (car set1) (intersection (cdr set1) set2)))
        (else (intersection (cdr set1) set2)) ))
```

`intersection` is $O(nm)$. (We usually say this is $O(n^2)$.

The implementation of `union` is exercise 2.59 in the text.

## 4.2 Representing sets as ordered lists

This can be done if the elements of which the sets are made have some natural order—for instance, if they are all integers. To make things simple, let us assume they are integers.

```
(define (element-of? x set)
  (cond ((null? set) #f)
        ((= x (car set)) #t)
        ((< x (car set)) #f)
        (else (element-of? x (cdr set))) ))
```

`element-of?` is $O(n)$ (but more like $n/2$ than $n$).

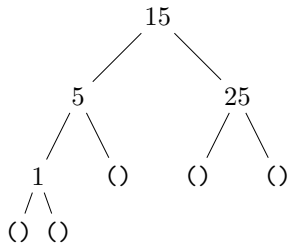
```
(define (intersection set1 set2)
  (if (or (null? set1) (null? set2))
      '()
      (let ((x1 (car set1))
            (x2 (car set2)) )
        (cond ((= x1 x2)
               (cons x1
                     (intersection (cdr set1) (cdr set2)) ))
              ((< x1 x2)
               (intersection (cdr set1) set2))
              ((> x1 x2)       ;;; could be ``else''
               (intersection set1 (cdr set2))) ))))
```

`intersection` is $O(n)$, rather than $O(n^2)$.

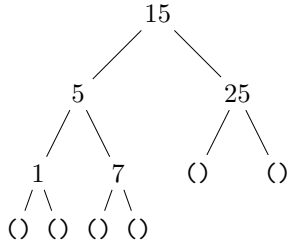
## 4.3 Representing sets as binary trees

This can also be done if the elements have a natural order. Again, let us assume all the elements are integers.

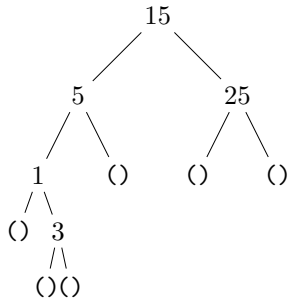
Suppose we start with the empty set and insert the numbers 15, 5, 25, and 1, in that order. The tree that gets constructed looks like this:




If we then insert (i.e., "adjoin") the number 7 to this set, we get




If instead of 7, we adjoin the number 3, we get




Here is how this is implemented: We will represent the tree as a recursive data structure: A tree consists of either

- an empty list (in which case the tree is empty), or
  - the root node,
  - the left subtree, and
  - the right subtree.

and each subtree is itself a data structure of the same form. (Thus, a leaf node of the tree is a node whose two subtrees are both empty lists.)

We can represent a non-empty tree therefore as a list of three elements

```
(entry left-branch right-branch)
```

where `entry` represents the root of the tree and either or both of the branches may be the empty list.

Of course this representation as a list is not the same structure as the tree we are thinking of, but this is just another example of the phenomenon we referred to before: the Scheme representation of a data structure does not have to "look" internally like the data structure as long as it can be made to act that way. To do this, we need a constructor and three selectors:

```
(define (make-tree entry left right)
  (list entry left right))
```

```
(define (entry tree) (car tree))            ;;; extracts first list element
(define (left-branch tree) (cadr tree))     ;;; extracts second list element
(define (right-branch tree) (caddr tree))   ;;; extracts third list element
```

In terms of this constructor and these selectors, we can now write

```
(define (element-of? x set)
  (cond ((null? set) #f)
        ((= x (entry set)) #t)
        ((< x (entry set))
         (element-of? x (left-branch set)))
        ((> x (entry set))       ;;; could be "else"
         (element-of? x (right-branch set))) ))
```

`element-of?` is $O(\log n)$ (if the tree is balanced).

```
(define (adjoin x set)
  (cond ((null? set) (make-tree x '() '()))
        ((= x (entry set)) set)
        ((< x (entry set))
         (make-tree (entry set)
                    (adjoin x (left-branch set))
                    (right-branch set)))
        ((> x (entry set))       ;;; could be "else"
         (make-tree (entry set)
                    (left-branch set)
                    (adjoin x (right-branch set)) ))))
```

`adjoin` is $O(\log n)$, similarly.