

Lists defined inductively

$LIST(A)$ is the **smallest** set satisfying this equation:

$$LIST(A) = \{ ' () \} \cup \{ (\mathbf{cons} \ a \ as) \mid a \in A, as \in LIST(A) \}$$

Equivalently, $LIST(A)$ is defined by these rules:

$$\frac{}{ ' () \in List(A) } \text{ (EMPTY)}$$

$$\frac{a \in A \quad as \in List(A)}{(\mathbf{cons} \ a \ as) \in List(A)} \text{ (CONS)}$$

One more inductive definition

A list of A is one of:

- **The empty list ' ()**
- **(cons a as), where a is an A and as is a list of A**

Lists generalized: S-expressions

An ordinary S-expression is one of:

- **An atom (symbol, number, Boolean)**
- **A list of ordinary S-expressions**

Can write literally in source, with quote

μ Scheme vs Impcore

New abstract syntax:

LET (keyword, names, expressions, body)

LAMBDA (formals, body)

APPLY (**exp**, actuals)

New concrete syntax for LITERAL:

(quote *S-expression*)

'*S-expression*

Equations and function for append

```
(append ' () ys) == ys
```

```
(append (cons z zs) ys) == (cons z (append zs ys))
```

```
(define append (xs ys)
```

```
  (if (null? xs)
```

```
      ys
```

```
      (cons (car xs) (append (cdr xs) ys))))
```

Naive list reversal

```
(define reverse (xs)
  (if (null? xs)
      '()
      (append (reverse (cdr xs))
               (list1 (car xs))))))
```

Reversal by accumulating parameters

```
(define revapp (xs ys)
  ; return (append (reverse xs) ys)
  (if (null? xs)
      ys
      (revapp (cdr xs)
              (cons (car xs) ys))))

(define reverse (xs) (revapp xs ' ()))
```

A-list example

```
-> (find 'Building  
      ' ((Course 105) (Building Barnum)  
        (Instructor Ramsey)))
```

Barnum

```
-> (val nr (bind 'Office 'Halligan-222  
              (bind 'Courses ' (105 150TW)  
                    (bind 'Email 'comp105-grades ' ())))))
```

```
((Email comp105-grades)  
 (Courses (105 150TW))  
 (Office Halligan-222))
```

```
-> (find 'Office nr)
```

Halligan-222

```
-> (find 'Favorite-food nr)  
( )
```


Laws of association lists

`(find k (bind k v a-1)) = v`

`(find k (bind k' v a-1)) = (find k a-1)`, provided `k != k'`

`(find k '()) = '()` --- bogus!