Pairs are the glue

- Using cons to build pairs, we can build data structures of unlimited complexity
- We can roll our own
  - if not too complex or if performance issues
- We can adopt a standard and use it for the basic elements of more complex structures
  - lists

Rational numbers with pairs

- An example of a fairly simple data structure that could be built directly with pairs

```
(define (make-rat n d)
  (cons n d))
```

```
(define (numer x)
  (car x))
```

```
(define (denom x)
  (cdr x))
```

```
(make-rat 1 2)
```

```
(cons 1 (cons 2 (cons nil nil)))
```
Extensibility

• What if we want to extend the data structure somehow?
• What if we want to define a structure that has more than two elements?
• We can use the pairs to glue pairs together in a more general fashion and so allow more general constructions
  » Lists

Fundamental list structure

• By convention, a list is a sequence of linked pairs
  » car of each pair is the data element
  » cdr of each pair points to list tail or the empty list

List construction

(define e (cons 1 (cons 2 (cons 3 '()))))

(define e (list 1 2 3))

procedure list

(list a b c ...)

• list returns a newly allocated list of its arguments
  » the arguments can be atomic items like numbers or quoted symbols
  » the arguments can be other lists
• The backbone structure of a list is always the same
  » a sequence of linked pairs, ending with a pointer to null (the empty list)
  » the car element of each pair is the list item
  » the list items can be other lists
List structure

(define a (list 4 5 6))
(define b (list 7 a 8))

Rational numbers with lists

(define (make-rat n d) (list n d))
(define (numer x) (car x))
(define (denom x) (cadr x))

Examples of list building

(cons 1 (cons 2 '()))
(cons 1 (list 2))
(list 1 2)

Lists and recursion

- A list is zero or more connected pairs
- Each node is a pair
- Thus the parts of a list (this pair, following pairs) are lists
- And so recursion is a natural way to express list operations
\textbf{cdr down}

- We can process each element in turn by processing the first element in the list, then recursively processing the rest of the list.

\begin{verbatim}
(define (length m)
  (if (null? m) 0 (+ 1 (length (cdr m)))))
\end{verbatim}

\textbf{cons up}

- We can build a list to return to the caller piece by piece as we go along through the input list.

\begin{verbatim}
(define (reverse m)
  (define (iter shrnk grow)
    (if (null? shrnk) grow
      (iter (cdr shrnk) (cons (car shrnk) grow))))
  (iter m '()))
\end{verbatim}

\textbf{sum the items in a list}

\begin{verbatim}
(sum the items in a list)
\end{verbatim}

\begin{verbatim}
(define (add-items m)
  (if (null? m) 0 (+ (car m) (add-items (cdr m))))

(+ 2 (+ 5 (+ 4 0)))
\end{verbatim}

\textbf{multiply each list element by 2}

\begin{verbatim}
(multiply each list element by 2)
\end{verbatim}

\begin{verbatim}
(define (double-all m)
  (if (null? m) '()
    (cons (* 2 (car m)) (double-all (cdr m))))

(double-all (list 4 0 -3))
\end{verbatim}
Variable number of arguments

- We can define a procedure that has zero or more required parameters, plus provision for a variable number of parameters to follow
  - The required parameters are named in the define statement as usual
  - They are followed by a "." and a single parameter name
- At runtime, the single parameter name will be given a list of all the remaining actual parameter values

(map)

- We can use the general purpose function map to map over the elements of a list and apply some function to them

```
(define (map p m)
  (if (null? m)
      '()
      (cons (p (car m))
            (map p (cdr m)))))

(define (double-all m)
  (map (lambda (x) (* 2 x)) m))
```