Lists

- Structures are great for situations when we know ahead of time what kind and how much data we will have.
  - can't add new stuff to a structure
  - have to name everything ahead of time
- Consider writing a program to sort some numbers.
- Lists allow us to have arbitrarily large collections of data.

cons

- We build (construct) lists with the cons operator:

```
(cons newitem list)
```

  - This creates a new list which has first item *newitem* and the rest of the list comes from *list*.
  - The empty list is specified as `empty`

new keyword!

```
(cons 'hello empty)
```

Making larger lists

```
(cons 1
   (cons 2
      (cons 3
         (cons 4 empty) ) ) )
```

Function that operates on a list

- Write a scheme function that adds up the three numbers in a list:

```
(add-3-numbers
   (cons 1
      (cons 2
         (cons 3))))
```

Extracting items from a list

`(first alist)`

is the first item in the list `alist`

`(rest alist)`

is everything except the first item in the list (all the rest of the items). This is always a list.

add-3-numbers

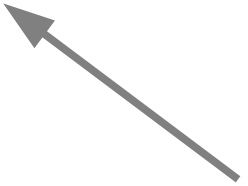
```
(define (add-3-numbers alist)
  (+ (first alist)
     (first (rest alist))
     (first (rest (rest alist)))))
```

Data Definition using lists - example

We could find the need for lists of things, for example lists of symbols:

A `list-of-symbols` is either

- the empty list, `empty`, or
- `(cons s los)` where `s` is a symbol and `los` is a list of symbols



Recursive(self-referential) data definition

Examples of *lists of symbols*

```
(cons 'Jelly empty)


(cons 'Dog
   (cons 'Cat
      (cons 'Goldfish
         (cons 'Python empty))))


empty
```

Exercise

- Create a *data definition* for a list of booleans.
  - the definition must cover all possible lists of boolean values.

Functions that operate on lists

- Write a function that determines whether a list of symbols contains the symbol `'Goldfish`.
  - function returns `true` or `false`

- The function must work with *any* list of symbols!

First try?

```
(define (contains-goldfish? l)
   (symbol=? 'Goldfish
      (first l)))
```

What if `'Goldfish` is the second item in the list?

Better ?

```
(define (contains-goldfish? l)
  (or
    (symbol=? 'Goldfish (first l))
    (symbol=? 'Goldfish (first (rest l)))))
```

What if the list has only one item?

What if the list is empty?

What if `'Goldfish` is the third item?

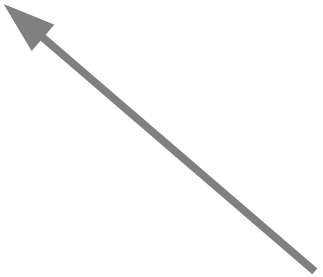
Define some tests

```
(contains-goldfish? empty)  => false
(contains-goldfish?
   (cons 'Goldfish empty)) => true
(contains-goldfish?
   (cons 'Fred empty)) => false

(contains-goldfish?
   (cons 'Fred
      (cons 'Goldfish empty))) => true
```

Check for empty list

```
(define (contains-goldfish? l)
  (cond
    [(empty? l) false]
    [ ...
```



is true only when l is empty list

What if it's not empty

```
(define (contains-goldfish? l)
  (cond
    [(empty? l) false]
    [else
      (cond
        [ ... deal with first ...]
        [ ... deal with rest ...]
```

Check first item

```
(define (contains-goldfish? l)
  (cond
    [(empty? l) false]
    [else
      (cond
        [(symbol=? 'Goldfish (first l)) true]
        [ ... deal with rest ...])
```

How do we deal with `(rest l)`?

- We need something that can tell us whether or not the symbol `'Goldfish` appears in the list of symbols `(rest l)`

- We could write a function that does this for us, call it `contains-goldfish2?`

Complete Function

```
(define (contains-goldfish? l)
  (cond
    [(empty? l) false]
    [else
      (cond
        [(symbol=? 'Goldfish (first l)) true]
        [else (contains-goldfish2? (rest l))]
       )]))
```

`contains-goldfish2?`

- We need a function that determines whether a list of symbols contains the symbol `'Goldfish`.
  - function returns `true` or `false`
- The function must work with *any* list of symbols!
- This should sound familiar!

Recursion

- `contains-goldfish?` and `contains-goldfish2?` do exactly the same thing.
  - we don't need two different functions.
  - we can just use `contains-goldfish?` for everything!

```
[ else (contains-goldfish? (rest l)) ]
```

Complete Function (*recursion)*

```
(define (contains-goldfish? l)
  (cond
    [(empty? l) false]
    [else
      (cond
        [(symbol=? 'Goldfish (first l)) true]
        [else (contains-goldfish? (rest l))]
       )]))
```

General Template for handling lists

```
(define (list-func alist)
  (cond
    [ (?empty alist) ... ]
    [ else ... (first alist) ...
           (rest alist) ...
```

Example: count items in a list

- We want a function that will count the number of items in a list (value of the function is a number).
- Call the function `how-many`

```
(how-many (cons 'a empty)) => 1
(how-many (cons 'fred
                (cons 'joe empty))) => 2
```

Apply our template

```
(define (how-many alist)
  (cond
    [ (?empty alist) 0 ]
    [ else ... (first alist) ...
           (rest alist) ...
```

What if the list is not empty?

- The number of elements in `alist` is:

  1 + number of elements in `(rest alist)`

```
(define (how-many alist)
  (cond
    [(empty? alist) 0]
    [else
      ( + 1 (how-many (rest alist)))]))
```

Exercise

- Write a function that will compute the sum of the elements of a list of numbers:

```
(sum empty) => 0

(sum (cons 1 empty)) => 1

(sum
  (cons 1
    (cons 2
      (cons 3 empty)))) => 6
```