Design Abstraction

- We deal with abstractions constantly:
  - we don't really know (or care) what mechanical/electrical operations take place when we issue the command "`(+ 3 5)`"
  - We have created structures to provide a level of abstraction:
    - circles, squares, shapes, etc. instead of x,y,radius,size,color, ...
  - We view a specific data structure as "a tree" and talk about (think about) "a tree" as if it actually existed...

Design

- One of the key concepts in design is recognition of situations that can benefit from abstraction:
  - simplification of how we express some computation.
  - simplification of how we implement some computation.
  - sometimes we can simplify the design process itself through abstraction.
    - designing recursive functions/data definitions is one example.

Similarities in Functions

- If we detect a high degree of similarity in the definition of two functions, this may be an opportunity to simplify things by writing a single (more abstract) function.
- Consider the following two functions:
  - `contains-doll?` determines whether a list of symbols contains the symbol `'doll`.
  - `contains-car?` determines whether a list of symbols contains the symbol `'car`.

contains-doll?

```
;; contains-doll? : los  ->  boolean
;; to determine whether alos contains
;; the symbol 'doll
(define (contains-doll? alos)
  (cond
    [(empty? alos) false]
    [else
      (cond
        [(symbol=? (first alos) 'doll) true]
        [else
          (contains-doll? (rest alos))])])))
```

contains-car?

```
;; contains-car? : los  ->  boolean
;; to determine whether alos contains
;; the symbol 'car
(define (contains-car? alos)
  (cond
    [(empty? alos) false]
    [else
      (cond
        [(symbol=? (first alos) 'car) true]
        [else
          (contains-car? (rest alos))])])))
```

More General Function

```
;; contains? : symbol los  ->  boolean
;; to determine whether alos contains the symbol s
(define (contains? s alos)
  (cond
    [(empty? alos) false]
    [else
       (cond
          [(symbol=? (first alos) s) true]
          [else
             (contains? s (rest alos))])]))
```

Using the more general function

- We can still write `contains-doll?` if we want:

```
(define (contains-doll? alos)
    (contains? 'doll alos))
```

- We can also not bother and just call `contains?` directly wherever we would have called `contains-doll?`

Another Example

- Functions we want:
  - `(below alon t)` constructs a list of numbers from alon that are less than t.
  - `(above alon t)` constructs a list of numbers from alon that are greater than t.
- Both can be considered as *filters* for a list
  - they *filter out* some elements.

below

```
;; below : lon number  ->  lon
;; construct a list of those numbers
;;   in alon that are below t
(define (below alon t)
  (cond
    [(empty? alon) empty]
    [else
      (cond
        [(< (first alon) t)
           (cons (first alon) (below (rest alon) t))]
        [else
              (below (rest alon) t)])]))
```

above

```
;; above : lon number  ->  lon
;; construct a list of those numbers
;;   in alon that are above t
(define (above above t)
  (cond
    [(empty? alon) empty]
    [else
      (cond
        [(> (first alon) t)
           (cons (first alon) (above (rest alon) t))]
        [else
              (above (rest alon) t)])]))
```

`above` and `below` in action

```
> (above '(8 1 7 253 49 26 4) 17)
(list 253 49 26)

> (below '(8 1 7 253 49 26 4) 17)
(list 8 1 7 4)
```

`above` and `below` Similarities

- Basically the same function, the only difference is the comparison operator.

- We can write a single function that determines what operator to use based on a parameter.
  - call the parameter `aorb` (`a`bove `or b`elow)

filter-above-or-below!

```
(define (filter-above-or-below alon t aorb)
  (cond
   [(empty? alon) empty]
   [else
         (cond
          [(symbol=? 'above aorb)
           (cond
            [(> (first alon) t)
                 (cons (first alon)
                            (filter-above-or-below (rest alon) t aorb))]
            [else
                 (filter-above-or-below (rest alon) t aorb)])]
          [(symbol=? 'below aorb)
           (cond
            [(< (first alon) t)
                 (cons (first alon)
                            (filter-above-or-below (rest alon) t aorb))]
            [else
                 (filter-above-or-below (rest alon) t aorb)])])]))
```

Function Generalization

- Abstraction can be useful, but as `filter-above-or-below` shows, it is not necessarily useful or desirable.
  - `filter-above-or-below` doesn't look much like either of the functions it replaces.
  - `filter-above-or-below` is harder to follow and doesn't make an obviously useful generalization (just does one of two possible operations).
  - there may be better generalizations possible...

A better approach

- We can build a useful generalization of the functions `above` and `below`
  - instead of passing a *flag* that indicates which comparison operator can be used, we pass an operator.
  - This is the kind of stuff scheme is really good for!
    - doing this kind of thing in other languages is much more complex.

Intermediate Student Grammer

```
<def>    =  (define (<var> <var> ...<var>) <exp>)
          | (define <var> <exp>)
          | (define-struct <var> (<var> <var> ...<var>))
<exp>    =    <var> | <boo> | <sym> | <prm> | empty
            | (<exp> <exp> ...<exp>)
            | (cond (<exp> <exp>) ...(<exp> <exp>))
            | (cond (<exp> <exp>) ...(else <exp>))
            | (local (<def> ...<def>) <exp>)
<var>    =    x | area-of-disk | circumference | ...
<boo>    =    true | false
<sym>    =    'a | 'doll | 'sum | ...
<num>    =    1 | -1 | 3/5 | 1.22 | ...
<prm>    =    + | - | cons | first | rest | ...
```

`filter1` function

```
(define (filter1 rel-op alon t)
  (cond
    [(empty? alon) empty]
    [else (cond
    [(rel-op (first alon) t)
     (cons (first alon)
           (filter1 rel-op (rest alon) t))]
    [else
     (filter1 rel-op (rest alon) t)])]))
```

`filter1` in action

```
> (filter1 < '(8 1 7 253 49 26 4) 17)
(list 8 1 7 4)

> (filter1 > '(8 1 7 253 49 26 4) 17)
(list 253 49 26)
```

Issues

- `filter1` looks like the functions it replaces (above and below).
- `filter1` is much more general:
  - filters out all elements that don't pass a relational test when compared with some number.
  - There are other possible uses for this function!

`filter1` can handle other operators

```
(filter1 >= '(1 14 27 19 33 15) 15)

(filter1 <= '(1 14 27 19 33 15) 15)

(filter1 = '(1 14 27 19 4 2 8) 8)
```

We can create our own operators!

```
;; is x a multiple of y (both integers) ?
(define (multiple? x y)
  (=
   (/ x y)
   (floor (/ x y))))

(filter1 multiple? '(4 18 17 25 30 88) 3) =>
'(18 30)
```

Operators and `filter1`

- Any function that has the following signature will work:

```
number number => boolean
```

- There are plenty of potential uses for `filter1`,
  - very good generalization.
  - a very nice abstraction.

Exercise

- Create a more general version of the `minimum` function shown below.
  - make sure it could also do "maximum"

```
(define (minimum alon)
  (cond
   [(empty? (rest alon)) (first alon)]
   [else
        (local
         ((define minrest (minimum (rest alon))))
         (cond
          [(< (first alon) minrest) (first alon)]
          [else minrest]))]))
```

Another Exercise

- Write a function named `filter2` that applies a unary boolean operator to each number in a list, to determine which numbers appear in the constructed list.
- Example usage:

```
(filter2 integer? '( 1 3.14 17 22.5 228 1.0001)) =>
'(1 17 228)
```

Exercise continued: Creating operators for `filter2`

- Create operators so that we can use filter2 to do the following:
  - extract all even numbers from a list.
  - extract all perfect squares (integer only) from a list.
  - extract all numbers less than 22 from a list
    - this is a clumsy way to do this – filter1 is much better, but this is possible with filter2

More `filter2` fun

- There should not be anything in the definition of `filter2` that requires that it only handle numbers.
- Consider using `filter2` to handle filtering a list of posn structures:
  - extract all those structures that have an x value of 0.
  - extract all those that have x value greater than y value.

Example `posn` operator for `filter2`

```
(define (posn-zero-x p)
  (= 0 (posn-x p)))


(filter2 posn-zero-x
   (list (make-posn 3 4)   (make-posn 0 5)
         (make-posn 0 200) (make-posn 200 0)
         (make-posn 0 0)   (make-posn 13 22))) =>


 (list (make-posn 0 5) (make-posn 0 200)
       (make-posn 0 0))
```

Extracting symbols from a list

- We need an operator for `filter2` that returns true if given a symbol.

```
(filter2 symbol?
    (list 1 3 'Hello (make-posn 2 3)
          3.14159 'World)) =>
(list 'Hello 'World)
```

`filter1` Exercise

- Go back to `filter1` and write a new operator that can be used by filter1 (and test it out):
  - extract all `posn` structures from a list that are more than 100 pixels from a specific `posn`.
    - for example, all the posn structures more than 100 pixels from the point 150,150

Another `filter1` exercise

- Extract all symbols from a list that are members of a set of symbols.
  - The set of symbols will become the parameter t

You need to write
this function

```
(filter1
   member?
   '(Hello world this is fun)
   '(Hi Hello Bonjour)) => '(Hello)
```