Abstraction

**Abstraction** is the process of finding similarities or common aspects, and forgetting unimportant differences.

Example: writing a function.

- The differences in parameter values are forgotten, and the similarity is captured in the function body.
- We have seen many similarities between functions, and captured them in function templates.

We already worked with `map`, `foldr`, `foldl`, and `filter`. So what’s new in this module?

Now that we have experience with recursion, we can look at these functions in a different way. We can see them as “abstractions” that capture similarities between code we could write recursively. All these things require that we understand how functions are first class values.

> Example: Eating apples

Consider two similar functions, `eat-apples` and `keep-odds`.

```scheme
(define (eat-apples lst)
  (cond [(empty? lst) empty]
        [(not (symbol=? (first lst) 'apple))
          (cons (first lst) (eat-apples (rest lst)))]
        [else (eat-apples (rest lst))]))
```
Consider two similar functions, `eat-apples` and `keep-odds`.

```scheme
(define (keep-odds lst)
  (cond [
    [(empty? lst) empty]
    [(odd? (first lst))
      (cons (first lst) (keep-odds (rest lst)))]
    [else (keep-odds (rest lst))]])
```

What these two functions have in common is their general structure. Where they differ is in the specific predicate used to decide whether an item is removed from the answer or not.

Because functions are first class values, we can write one function to do both these tasks because we can supply the predicate to be used as an argument to that function.

The built-in function `filter`, of course, is exactly what we seek. How can we write our own?

```scheme
(define (my-filter pred? lst)
  (cond [[(empty? lst) empty]
    [(pred? (first lst))
      (cons (first lst) (my-filter pred? (rest lst)))]
    [else (my-filter pred? (rest lst))]])
```

> Example: Keeping odd numbers

> Example: Abstracting out differences

> Abstracting `keep-odds` to `my-filter`
Tracing my-filter

(define (my-filter pred? lst)
  (cond [(empty? lst) empty]
        [(pred? (first lst))
         (cons (first lst) (my-filter pred? (rest lst)))]
        [else (my-filter pred? (rest lst))])

(my-filter even? (list 0 1 2 3 4))
⇒ (cond [(empty? (list 0 1 2 3 4)) empty]
        [(even? (first (list 0 1 2 3 4)))
         (cons (first (list 0 1 2 3 4))
               (my-filter even? (rest (list 0 1 2 3 4)))]
        [else (my-filter even? (rest (list 0 1 2 3 4)))]
        ⇒ (cons 0 (my-filter even? (list 1 2 3 4)))
        ⇒ (cons 0 (my-filter even? (list 2 3 4)))
        ⇒ (cons 0 (cons 2 (cons 4 empty)))

filter

my-filter performs the same actions as the built-in function filter.

filter handles the general operation of selectively keeping items on a list.

Functions such as filter that consume a (listof X) and a function to generalize it are called abstract list functions at the University of Waterloo.

The class of functions that either consume or produce functions is called higher order functions. Higher order functions include filter, map, foldr, etc., and also functions that produce functions, such as make-adder.

map

Recall: map consumes a function and a list, and transforms items in the list with the function.
(map sqr (list 2 3 5 7 11)) ⇒ (list 4 9 25 49 121)
(map add1 (list 2 3 5 7 11)) ⇒ (list 3 4 6 8 12)

Here are two functions that each do something “map-like”.
(define (sqr-each lst)
  (cond [(empty? lst) empty]
        [else (cons (sqr (first lst)) (sqr-each (rest lst)))]))

(define (add1-each lst)
  (cond [(empty? lst) empty]
        [else (cons (add1 (first lst)) (add1-each (rest lst)))]))

Abstract out the differences between sqr-each and add1-each to write a function my-map that consumes a function and a (listof Any), and behaves like map.

(my-map sqr (list 2 3 5 7 11)) ⇒ (list 4 9 25 49 121)
(my-map add1 (list 2 3 5 7 11)) ⇒ (list 3 4 6 8 12)
**Functional abstraction** is the process of creating abstract functions such as `filter`. Advantages include:

1. Reducing code size.
3. Fixing bugs in one place instead of many.
4. Improving one functional abstraction improves many applications.

We will do more of this in the next lecture module.

---

**Example: character transformation in strings**

Suppose during a computation, we want to specify some action to be performed one or more times in the future.

Before knowing about `lambda`, we might build a data structure to hold a description of that action, and a helper function to consume that data structure and perform the action.

Now, we can just describe the computation clearly using `lambda`.

---

We'd like a function, `transform`, that transforms one string into another according to a set of rules that are specified when it is applied.

In one application, we might want to change every instance of ‘a’ to a ‘b’. In another, we might transform lowercase characters to the equivalent uppercase character and digits to ‘*’.  

```scheme
(check-expect (transform "abracadabra" ...) "bbrcbdbbrb")
(check-expect (transform "Testing 1-2-3" ...) "TESTING *-*-*")
```

We use `...` to indicate that we still need to supply some arguments.
We could imagine `transform` containing a `cond`:

```
(cond [(char=? ch \a) \b]
    [(char-lower-case? ch) (char-upcase ch)]
    [(char-numeric? ch) \*]
    ...)
```

But this fails for a number of reasons:

- The rules are “hard-coded”; we want to supply them when `transform` is applied.
- A lower case ‘a’ would always be transformed to ‘b’; never to ‘B’.

But the idea is inspiring...

Suppose we supplied `transform` with a list of question/answer pairs:

```
;; A TransformSpec is one of:
;; * empty
;; * (cons (list Question Answer) TransformSpec)
```

Like `cond`, we could work our way through the `TransformSpec` with each character. If the Question produces `true`, then apply the Answer to the character. If the Question produces `false`, go on to the next Question/Answer pair.

What are the types for `Question` and `Answer`?

Functions as first class values can help us. Both `Question` and `Answer` are functions that consume a `Char`.

`Question` produces a `Bool` and `Answer` produces a `Char`. This completes our data definition, above:

```
;; A Question is a (Char -> Bool)
;; An Answer is a (Char -> Char)
```

And a completed example:

```
(check-expect (transform "Testing 1-2-3"
    (list (list char-lower-case? char-upcase)
           (list char-numeric? (lambda (ch) \*))))
"TESTING *-*-*")
```
transform consumes a string and produces a string but we need to operate on characters. This suggests a wrapper function:

```scheme
;; A TransformSpec is one of:
;; * empty
;; * (cons (list Question Answer) TransformSpec)
```

```scheme
;; (transform s spec) transforms the string s according to the given specification.
;; transform: Str TransformSpec -> Str
(define (transform s spec)
  (list->string (trans-loc (string->list s) spec)))
```

```scheme
;; trans-loc (listof Char) TransformSpec -> (listof Char)
(check-expect (trans-loc (list #\a #\9) (list (list char-lower-case? char-upcase)))
  (list #\A #\9))
(define (trans-loc loc spec)
  (cond 
        [(empty? loc) empty]
        [(cons? loc) (cons (trans-char (first loc) spec)
                          (trans-loc (rest loc) spec))]
  )

(define (trans-char ch spec)
  (cond 
        [(empty? spec) ch]
        [((first (first spec)) ch) ((second (first spec)) ch)]
        [else (trans-char ch (rest spec))])

(check-expect (transform "Testing 1-2-3"
               (list (list char-lower-case? char-upcase)
                     (list char-numeric? (lambda (ch) #\*))))
  "TESTING *-*-*")
(check-expect (transform "abracadabra"
               (list (lambda (ch) (char=? ch #\a))
                     (lambda (ch) #\b)))
  "bbrbcdbbbbrb")
```

The repeated lambda expressions suggest some utility functions:
```
(define (is-char? c1) (lambda (c2) (char=? c1 c2)))
(define (always c1) (lambda (c2) c1))
```
Here are two simple list functions:

```scheme
(define (negate-list lst)
  (cond [(empty? lst) empty]
        [else (cons (- (first lst))
                  (negate-list (rest lst)))]))

(define (compute-taxes payroll)
  (cond [(empty? payroll) empty]
        [else (cons (sr->tr (first payroll))
                   (compute-taxes (rest payroll)))]))
```

> Abstracting another set of examples

We look for a difference that can't be explained by renaming (it being what is applied to the first item of a list) and make that a parameter.

```scheme
(define (negate-list lst)
  (cond [(empty? lst) empty]
        [else (cons (- (first lst))
                  (negate-list (rest lst)))]))

(define (compute-taxes payroll)
  (cond [(empty? payroll) empty]
        [else (cons (sr->tr (first payroll))
                   (compute-taxes (rest payroll)))]))

(define (my-map f lst)
  (cond [(empty? lst) empty]
        [else (cons (f (first lst))
                   (my-map f (rest lst)))]))
```

> Tracing my-map

```scheme
(my-map (my-map f list) lst)
  (cond [(empty? lst) empty]
        [else (cons (f (first lst))
                   (my-map f (rest lst)))]))
```

```
⇒ (cons 9 (my-map sqr (list 3 6 5)))
⇒ (cons 9 (cons 36 (my-map sqr (list 6 5))))
⇒ (cons 9 (cons 36 (cons 25 (my-map sqr empty))))
⇒ (cons 9 (cons 36 (cons 25 empty)))
```

my-map performs the general operation of transforming a list element-by-element into another list of the same length.
> Effect of my-map

\[(\text{my-map } f \ (\text{list } x_1 \ x_2 \ \ldots \ x_n))\] has the same effect as evaluating \[(\text{list } (f \ x_1) \ (f \ x_2) \ \ldots \ (f \ x_n)).\]

\[(\text{my-map even? } (\text{list } 0 \ 1 \ 2 \ 3 \ 4))\]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>even?</td>
<td>even?</td>
<td>even?</td>
<td>even?</td>
<td>even?</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>

> Using my-map

We can use `my-map` to give short definitions of a number of functions we have written to consume lists:

- `(define (negate-list lst) (my-map - lst))`
- `(define (compute-taxes lst) (my-map sr->tr lst))`

How can we use `my-map` to rewrite `trans-loc`?

> The contract for my-map

`my-map` Consumes a function and a list, and produces a list.

How can we be more precise about its contract, using parametric type variables?
In addition to `filter`, Intermediate Student also provides `map` as a built-in function, as well as many other higher order functions. Check out the Help Desk (in DrRacket, Help → Help Desk → How to Design Programs Languages → 4.17 Higher-Order Functions)

The functions `map` and `filter` allow us to quickly describe functions to do something to all elements of a list, and to pick out selected elements of a list, respectively.

The functions we have worked with so far consume and produce lists.

What about abstracting from functions such as `count-symbols` and `sum-of-numbers`, which consume lists and produce simple values?

Let's look at these, find common aspects, and then try to generalize from the template.

---

```scheme
(define (sum-of-numbers lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst)
                  (sum-of-numbers (rest lst)))]))

(define (prod-of-numbers lst)
  (cond [(empty? lst) 1]
        [else (* (first lst)
                 (prod-of-numbers (rest lst)))]))

(define (all-true? lst)
  (cond [(empty? lst) true]
        [else (and (first lst)
                    (all-true? (rest lst)))]))
```
Note that each of these examples has a base case which is a value to be returned when the argument list is empty.

Each example is applying some function to combine \((\text{first \ lst})\) and the result of a recursive function application with argument \((\text{rest \ lst})\).

This continues to be true when we look at the list template and generalize from that.

---

We replace the first ellipsis by a base value.

We replace the rest of the ellipses by some function which combines \((\text{first \ lst})\) and the result of a recursive function application on \((\text{rest \ lst})\).

This suggests passing the base value and the combining function as parameters to an abstract list function.

---

We could write our own \texttt{foldr}:

\begin{verbatim}
(define (my-foldr combine base lst)
  (cond [(empty? lst) base]
        [else (combine (first lst) ...
                    (my-foldr combine base (rest lst)))])
)
\end{verbatim}

\texttt{foldr} is also a built-in function in Intermediate Student with lambda.
(define (my-foldr combine base lst)
  (cond [(empty? lst) base]
        [else (combine (first lst)
                        (my-foldr combine base (rest lst)))]))

(my-foldr + 0 '(1 2 3 4 5))

(foldr string-append "2B" (list "To" "be" "or" "not")) ⇒ "Tobenot2B"

(foldr string-append "2B" (list "To" "be" "or" "not")) ⇒ "Tobenot2B"

(foldr + 0 '(1 2 3 4 5))

(foldr + 0 '(1 2 3 4 5))

foldr is short for “fold right”.

The reason for the name is that it can be viewed as “folding” a list using the provided combine function, starting from the right-hand end of the list.

Foldr can be used to implement map, filter, and other abstract list functions.
The contract for `foldr`

`foldr` consumes three arguments:

- a function which combines the first list item with the result of reducing the rest of the list;
- a base value;
- a list on which to operate.

What is the contract for `foldr`?

Aside: comparison to imperative languages

Imperative languages, which tend to provide inadequate support for recursion, usually provide looping constructs such as “while” and “for” to perform repetitive actions on data.

Abstract list functions cover many of the common uses of such looping constructs.

Our implementation of these functions is not difficult to understand, and we can write more if needed, but the set of looping constructs in a conventional language is fixed.

Summary: `foldr` vs. the list template

Anything that can be done with the list template can be done using `foldr`, without writing any recursive code.

Does that mean that the list template is obsolete?

No. Experienced Racket programmers still use the list template, for reasons of readability and maintainability.

Abstract list functions should be used judiciously, to replace relatively simple uses of recursion.
Generalizing accumulative recursion

Let's look at several past functions that use recursion on a list with one accumulator.

;; code from lecture module 12

(define (sum-list lst0)
  (local [define (sum-list/acc lst sum-so-far)
               (cond [(empty? lst) sum-so-far]
                     [else (sum-list/acc (rest lst)
                                          (+ (first lst) sum-so-far))])]
        (sum-list/acc lst0 0)))

(check-expect (sum-list (list 1 2 3 4)) 10)

> Generalizing accumulative recursion

Let's look at several past functions that use recursion on a list with one accumulator.

;; code from lecture module 9 rewritten to use local

(define (rev-list lst0)
  (local [(define (rev-list/acc lst lst-so-far)
               (cond [(empty? lst) lst-so-far]
                     [else (rev-list/acc (rest lst)
                                          (cons (first lst) lst-so-far))])]
        (rev-list/acc lst0 empty)))

(check-expect (rev-list (list 1 2 3 4 5)) (list 5 4 3 2 1))

> foldl

The differences between these two functions are:

- the initial value of the accumulator;
- the computation of the new value of the accumulator, given the old value of the accumulator and the first element of the list.
> **foldl**

(define (my-foldl combine base lst0)
  (local [(define (foldl/acc lst acc)
            (cond [(empty? lst) acc]
                  [else (foldl/acc (rest lst) (combine (first lst) acc))]]))
    (foldl/acc lst0 base)))

(define (sum-list lon) (my-foldl + 0 lon))
(define (my-reverse lst) (my-foldl cons empty lst))

foldl is defined in the Intermediate Student language and above.

> **foldl**

We noted earlier that intuitively, the effect of the application
(foldr f b (list x_1 x_2 ... x_n))
is to compute the value of the expression
(f x_1 (f x_2 (... (f x_n b) ...)))

What is the intuitive effect of the following application of foldl?
(foldl f b (list x_1 ... x_n-1 x_n))

> Tracing foldl

(foldl f b (list x_1 x_2 ... x_n)) (f x_n (f x_n-1 (... (f x_1 b))))
(foldl string-append "2B" (list "To" "be" "or" "not")) ⇒ "notorbeTo2B"

```
string-append
  "2B"    "To2B" "beTo2B" "orbeTo2B" "notorbeTo2B"
```

```
To
To2B
```

```
be
beTo2B
```

```
or
orbeTo2B
```

```
not
notorbeTo2B
```
foldl is short for “fold left”.

The reason for the name is that it can be viewed as “folding” a list using the provided combine function, starting from the left-hand end of the list.

> Contract for foldl

What is the contract of foldl?

Exercise

Manually evaluate the two expressions:

(foldl (lambda (x y) (+ x y y)) 1 (list 3 4 5))

(foldr (lambda (x y) (+ x y y)) 1 (list 3 4 5))

Are the values the same? Why or why not?

Then check your answer using DrRacket.
Goals of this module

- You should be able to produce functions using \texttt{lambda}.
- You should understand how \texttt{lambda} underlies our usual definition of functions.
- You should be familiar with the built-in functions \texttt{filter}, \texttt{map}, \texttt{foldr}, and \texttt{foldl}. You should understand how they abstract common recursive patterns, and be able to use them to write code.
- You should be able to write your own abstract list functions that implement other recursive patterns.
- You should understand how to do step-by-step evaluation of programs written in the Intermediate language that make use of functions as values.

Further Reading: HtDP, sections 21-24.

Summary: built-in functions

In this module we added the following to our toolbox:

These are the functions and special forms currently in our toolbox: