Anonymous functions
Anonymous functions

(define (make-adder n)
  (local
    [(define (f m) (+ n m))]
    f))

The result of evaluating this expression is a function.

What is its name? It is anonymous (has no name).

This is sufficiently valuable that there is a special mechanism for it.
Producing anonymous functions

\[
\begin{align*}
&\text{(define (not-an-apple? item)}
&\quad (\text{not (symbol=? item 'apple)})) \\
&(\text{define (eat-apples lst)}
&\quad (\text{filter not-an-apple? lst}))
\end{align*}
\]

This is a little unsatisfying, because not-an-apple? is such a small and relatively useless function.

It is unlikely to be needed elsewhere.

We can avoid cluttering the top level with such definitions by putting them in \textbf{local} expressions.
(define (eat-apples lst)
  (local
   [(define (not-an-apple? item)
     (not (symbol=? item 'apple)))]
   (filter not-an-apple? lst)))

This is as far as we would go based on our experience with `local`.

But now that we can use functions as values, the value produced by the local expression can be the function `not-an-apple?`.

We can then take that value and deliver it as an argument to `filter`. 
(define (eat-apples lst)
  (filter
    (local
      [(define (not-an-apple? item)
          [(define (not (symbol=? item 'apple))]
            not-symbol-apple?)
      lst))
)

But this is still unsatisfying. Why should we have to name not-an-apple? at all? In the expression (* (+ 2 3) 4), we did not have to name the intermediate value 5.

Racket provides a mechanism for constructing a nameless function which can then be used as an argument.
Introducing **lambda**

(local
   [(define (name-used-once x1 ... xn) exp)]
   name-used-once)

can also be written (**lambda** (x1 ... xn)exp)

**lambda** can be thought of as “make-function”.

It can be used to create a function which we can then use as a value, for example, as the value of the first argument of **filter**.
We can then replace

\[
(\text{define (eat-apples lst)}
  \text{ (filter (local}
    \text{ [(define (not-an-apple? item)
      \text{ (not (symbol=? item 'apple)))]
    \text{ not-an-apple?)
  \text{ lst}))}
\]

with the following:

\[
(\text{define (eat-apples lst)}
  \text{ (filter (lambda (item) (not (symbol=? item 'apple)))
  lst))}
\]
**lambda** is available in Intermediate Student with Lambda, and discussed in section 24 of the textbook.

We are jumping ahead to it because of its central importance in Racket, Lisp, and the history of computation in general.

The designers of the teaching languages could have renamed it as they did with other constructs, but chose not to out of respect.

The word **lambda** comes from the Greek letter \( \lambda \), used as notation in the first formal model of computation.
We can use `lambda` to simplify `make-adder`. Instead of

```
(define (make-adder n)
  (local
    [(define (f m) (+ n m))]
    f))
```

we can write:

```
(define (make-adder n)
  (lambda (m) (+ n m)))
```
lambda also underlies the definition of functions.

Until now, we have had two different types of definitions.

;; a definition of a numerical constant
(define interest-rate 0.03)

;; a definition of a function to compute interest
(define (interest-earned amount)
  (* interest-rate amount))

There is really only one kind of define, which binds a name to a value.
Internally,

\[(\text{define} \ (\text{interest-earned} \ \text{amount})
\quad (\ast \ \text{interest-rate} \ \text{amount}))\]

is translated to

\[(\text{define} \ \text{interest-earned}
\quad (\text{lambda} \ (\text{amount}) \ (\ast \ \text{interest-rate} \ \text{amount})))\]

which binds the name interest-earned to the value

\[(\text{lambda} \ (\text{amount}) \ (\ast \ \text{interest-rate} \ \text{amount})).\]
We should change our semantics for function definition to represent this rewriting.

But doing so would make traces much harder to understand.

As long as the value of defined constants (now including functions) cannot be changed, we can leave their names unsubstituted in our traces for clarity.

In stepper questions, if a function is defined using function syntax, you can skip the lambda substitution step. If a function is defined as a constant using lambda, you must include the lambda step.
For example, here is make-adder rewritten using lambda.

\[
\text{(define make-adder (lambda (x) (lambda (y) (+ x y))))}
\]

What is ((make-adder 3) 4)?
(define make-adder (lambda (x) (lambda (y) (+ x y))))
((make-adder 3) 4)) ;; substitute the λ expression
=> (((lambda (x) (lambda (y) (+ x y))) 3) 4))
=> ((lambda (y) (+ 3 y)) 4))
=> (+ 3 4))
=> 7

make-adder is defined as a constant using lambda, so it is substituted in place of make-adder
Syntax and semantics
Syntax and semantics of Intermediate Student w/ lambda

**Before**

First position in an application must be a built-in or user-defined function.

A function name had to follow an open parenthesis.

**Now**

First position can be an expression (computing the function to be applied). Evaluate it along with the other arguments.

A function application can have two or more open parentheses in a row: ((make-adder 3) 4).
We need a rule for evaluating applications where the function being applied is anonymous (a lambda expression.)

\[
((\text{lambda} \ (x_1 \ldots \ x_n) \ \text{exp}) \ v_1 \ldots \ v_n))
\]

\[\Rightarrow \ \text{exp}'\]

where exp' is exp with all occurrences of \(x_1\) replaced by \(v_1\), all occurrences of \(x_2\) replaced by \(v_1\), and so on.

As an example:

\[
((\text{lambda} \ (x \ y) \ ((\ast \ (+ \ y \ 4) \ x)) \ 5 \ 6))
\]

\[\Rightarrow ((\ast \ (+ \ 6 \ 4) \ 5))\]
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`.
Example: character translation in strings

We would like a function, translate, that translates 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 translate lowercase characters to the equivalent uppercase character and digits to '*'.

(check-expect
   (translate "abracadabra" ...) "bbrbcdbbbrb")
(check-expect
   (translate "Testing 1-2-3" ...) "TESTING *-*-*")

We use ... to indicate that we still need to supply some arguments.
We could imagine translate 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 translate is applied.
• A lower case 'a' would always be translated to 'b'; never to ‘B’

But the idea is inspiring...
Suppose we supplied translate with a list of question / answer pairs:

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

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

What are the types for (list Question 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 character. This completes our data definition, above:

;; A Question is a Char -> Bool
;; An Answer is a Char -> Char

And a completed example:

(check-expect
 (translate "Testing 1-2-3"
  (list (list char-lower-case? char-upcase)
       (list char-numeric? (lambda (ch) #\*/))))
 "TESTING *--*-"
Translate: developing the code

translate consumes a string and produces a string but we need to operate on characters. This suggests a wrapper function:

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

;; (translate s spec) translates the string s according to the given specification
;; translate: Str TranslateSpec -> Str
(define (translate s spec)
  (list->string (trans-loc (string->list s) spec)))
;;; trans-loc:  
;;; (listof Char) TranslateSpec -> (listof Char)
(check-expect  
  (trans-loc (list #\a #\9)  
    (list (list char-lower-case? char-upcase)))) (list#  
(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
  (translate "Testing 1-2-3"
    (list (list char-lower-case? char-upcase)
      (list char-numeric? (lambda (ch) #\*))))
  "TESTING *-*-*")

(check-expect
  (translate "abracadabra"
    (list (list (lambda (ch) (char=? (lambda (ch) #\b))))))
  "bbrbcbdbbrb")

The repeated lambda expressions suggest some utility functions:
(define (is-char? c1) (lambda (c2) (char=? c1 c2)))
(define (always c1) (lambda (c2) c1))
Abstracting from examples
Abstracting another set of examples

Here are two early list functions we wrote.

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

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

(my-map \texttt{sqr} '(3 6 5))
=> (\texttt{cons} 9 (my-map \texttt{sqr} '(6 5)))
=> (\texttt{cons} 9 (\texttt{cons} 36 (my-map \texttt{sqr} '(5))))
=> (\texttt{cons} 9 (\texttt{cons} 36 (\texttt{cons} 25 (my-map \texttt{sqr} '()))))
=> (\texttt{cons} 9 (\texttt{cons} 36 (\texttt{cons} 25 \texttt{empty})))

my-map performs the general operation of transforming a list element-by-element into another list of the same length.
The application (my-map f (list x1 x2 ... xn)) has the same effect as evaluating (list (f x1) (f x2) ... (f xn)).

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?
Built-in abstract list functions

Intermediate Student also provides `map` as a built-in function, as well as many other abstract list functions. Check out the Help Desk (in DrRacket, Help → Help Desk → How to Design Programs Languages → 4.17 Higher-Order Functions)

The abstract list 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.
Abstracting another set of examples

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 values?

Let us look at these, find common aspects, and then try to generalize from the template.
(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 (count-symbols lst)
  (cond
    [(empty? lst) 0]
    [else (+ 1 (count-symbols (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 (first lst) and the result of a recursive function application with argument (rest lst).

This continues to be true when we look at the list template and generalize from that.
(define (list-template lst)
  (cond
    [(empty? lst) ...]
    [else (...
      ...(first lst)
      ...(list-template (rest lst)))]))

We replace the first ellipsis by a base value.

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

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

```
(define (my-foldr combine base lst)
  (cond
   [(empty? lst) base]
   [else
    (combine (first lst)
     (my-foldr combine base (rest lst)))]))
```

**foldr** is also a built-in function in Intermediate Student With Lambda.
Tracing my-foldr

(my-foldr f 0 '(3 6 5))
=> (f 3 (my-foldr f 0 '(6 5)))
=> (f 3 (f 6 (my-foldr f 0 '(5))))
=> (f 3 (f 6 (f 5 (my-foldr f 0 '()))))
=> (f 3 (f 6 (f 5 0)))
=> ...

Intuitively, the effect of the application
(foldr f b (list x1 x2 ... xn))
is to compute the value of the expression
(f x1 (f x2 (... (f xn b) ...))).
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, and
• a list on which to operate.

What is the contract for foldr?
Using **foldr**

\[
(\text{define } (\text{sum-of-numbers } \text{lst}) (\text{foldr } + \ 0 \ \text{lst}))
\]

If \( \text{lst} \) is \((\text{list } x_1 x_2 \ldots x_n)\), then by our intuitive explanation of **foldr**, the expression \((\text{foldr } + \ 0 \ \text{lst})\) reduces to

\[
( + \ x_1 ( + \ x_2 ( + \ldots ( + \ x_n \ 0) \ldots)))
\]

Thus **foldr** does all the work of the template for processing lists, in the case of **sum-of-numbers**.
The function provided to \texttt{foldr} consumes two parameters: one is an element on the list which is an argument to \texttt{foldr}, and one is the result of reducing the rest of the list.

Sometimes one of those arguments should be ignored, as in the case of using \texttt{foldr} to compute count-symbols.
The important thing about the first argument to the function provided to `foldr` is that it contributes 1 to the count; its actual value is irrelevant.

Thus the function provided to `foldr` in this case can ignore the value of the first parameter, and just add 1 to the reduction of the rest of the list.
(define (count-symbols lst)
  (foldr (lambda (x rror) (add1 rror)) 0 lst))

The function provided to foldr, namely (lambda (x rror) (add1 rror)), ignores its first argument.

Its second argument is the result of recursing on the rest (rror) of the list (in this case the length of the rest of the list, to which 1 must be added).
More examples

What do these functions do?

\[
\text{(define (bar lon)}
\quad \text{(foldr max (first lon) (rest lon)))}
\text{)}
\]
\[
\text{(bar '(1 5 23 3 99 2))}
\]

\[
\text{(define (foo los)}
\quad \text{(foldr)}
\quad \text{\quad (lambda (s rror) (+ (string-length s) rror))}
\quad \text{0}
\quad \text{los))}
\text{)}
\]
\[
\text{(foo '("one" "two" "three")})
\]

Using foldr to produce lists

So far, the functions we have been providing to foldr have produced numerical results, but they can also produce cons expressions.

foldr is an abstraction of structural recursion on lists, so we should be able to use it to implement negate-list from module 05.

We need to define a function (lambda (xrror) ...) where x is the first element of the list and rror is the result of the recursive function application.

negate-list takes this element, negates it, and cons it onto the result of the recursive function application.
The function we need is

\[(\text{lambda} \ (x \ \text{rror}) \ (\text{cons} \ (- \ x) \ \text{rror}))\]

Thus we can give a non-recursive version of negate-list (that is, \textit{foldr} does all the recursion).

\[(\text{define} \ (\text{negate-list} \ \text{lst}) \n \quad (\text{foldr} \ (\text{lambda} \ (x \ \text{rror}) \ (\text{cons} \ (- \ x) \ \text{rror})) \n \quad \text{empty} \n \quad \text{lst}))\]

Because we generalized negate-list to \textit{map}, we should be able to use \textit{foldr} to define \textit{map}. 
Let us look at the code for my-map.

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

Clearly empty is the base value, and the function provided to foldr is something involving cons and f.
In particular, the function provided to \texttt{foldr} must apply \( f \) to its first argument, then \texttt{cons} the result onto its second argument (the reduced rest of the list).

\[
\text{(define (my-map } f \text{ lst)} \quad  \\
\text{ (foldr (lambda (x rror) (cons (f x) rror))}} \quad  \\
\text{ empty lst))}
\]

We can also implement my-filter using \texttt{foldr}. 
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.
Anything that can be done with the list template can be done using foldr, without explicit recursion (unless it ends the recursion early, like insert).

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

Lets look at several past functions that use recursion on a list with one accumulator.

;;; code from lecture module 12
(define (sum-list lon)
  (local
    [(define (sum-list/acc lon0 sum-so-far)
        (cond
          [(empty? lon0) sum-so-far]
          [else (sum-list/acc (rest lon0)
                              (+ (first lon0) sum-so-far))]]))
    (sum-list/acc lon 0)))
;; code from lecture module 9 rewritten to use local
(define (my-reverse lst)
  (local
   [(define (my-rev/acc lst0 list-so-far)
     (cond
      [(empty? lst0) list-so-far]
      [else (my-rev/acc (rest lst0)
                          (cons (first lst0) list-so-far))]]))
  (my-rev/acc lst empty)))
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.
(define (my-foldl1 combine base lst)
  (local
    [(define (foldl/acc lst0 acc)
        (cond
          [(empty? lst0) acc]
          [else (foldl/acc (rest lst0)
                          (combine (first lst0) acc))])]]
    (foldl/acc lst base)))

(define (sum-list lon) (my-foldl1 + 0 lon))
(define (my-reverse lst) (my-foldl1 cons empty lst))
We noted earlier that intuitively, the effect of the application

\( \text{foldr } f \ b \ '(x_1 \ x_2 \ldots \ x_n) \)

is to compute the value of the expression

\( (f \ x \ 1 \ (f \ x \ 2 \ (\ldots \ (f \ x \ n \ b) \ \ldots))) \)

What is the intuitive effect of the following application of \text{foldl}?

\( \text{foldl } f \ b \ '(x_1 \ldots \ x_{n-1} \ x_n) \)

The function \text{foldl} is provided in Intermediate Student. What is the contract of \text{foldl}?
Higher-order functions
Higher-order functions

Functions that consume or produce functions like `filter`, `map`, and `foldr` are sometimes called *higher-order functions*.

Another example is the built-in `build-list`. This consumes a natural number `n` and a function `f`, and produces the list

```
(list (f 0) (f 1) ... (f (sub1 n)))
```

```
(build-list 4 (lambda (x) x))
```

=> `'(0 1 2 3).

Clearly `build-list` abstracts the “count up” pattern, and it is easy to write our own version.
(define (my-build-list n f)
  (local
    [(define (list-from i)
        (cond
          [((>= i n) empty]
          [else (cons (f i) (list-from (add1 i)))]))
    (list-from 0)))
Build-list examples

$$
\sum_{i=0}^{n-1} x_i
$$

(define (sum n f)
  (foldr + 0 (build-list n f)))

(sum 4 sqr)

=> 14
We can now simplify mult-table even further.

```
(define (mult-table nr nc)
 (build-list nr
   (lambda (r)
      (build-list nc
        (lambda (c)
          (* r c)))))
)```
The following partially completed Racket function should convert a string to uppercase:

```
(define (string-upcase s)
  (local
    [(define old-list (string->list s))
     (define new-list (... HERE ...))
     (define new-string (list->string new-list))]
    new-string))
```

Which would be the most appropriate abstract list function to combine with `char-upcase` to fill in the code at HERE?

A) `filter`  B) `map`  C) `build-list`  D) `lambda`  E) `append`
Goals of this module

• You should understand the idea of functions as first-class values: how they can be supplied as arguments, produced as values using `lambda`, bound to identifiers, and placed in lists.

• You should be familiar with the built-in abstract list functions provided by Racket, 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.