15: Lambda

Anonymous Functions

Functions as first-class values (previous module) offer many new possibilities for writing programs, including:

- Functions that consume functions as arguments, modifying their behaviour.
- Functions that produce new functions, customized by the first function's parameters.
- Functions bound to constants for later use.
- Functions stored in data structures, also for later use.

Lambda provides a clean, lean way to produce a function that makes it easier to use functions as first-class values (last module) and work exceptional well with higher-order functions (next module).

Anonymous functions

```
(define (make-adder n)
  (local [(define (f m) (+ n m))]
    f))
(make-adder 3)
```

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.

```
(define (not-symbol-apple? item) (not (symbol=? item 'apple)))
(define (eat-apples lst) (filter not-symbol-apple? lst))
```

This is a little unsatisfying, because `not-symbol-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 **`local`** expressions.

```
(define (eat-apples lst)
  (local [(define (not-symbol-apple? item)
            (not (symbol=? item 'apple)))]
    (filter not-symbol-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-symbol-apple?`.

We can give that value as an argument to `filter`.

```
(define (eat-apples lst)
  (filter (local [(define (not-symbol-apple? item)
                    (not (symbol=? item 'apple)))]
            not-symbol-apple?)
          lst))
```

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

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

```
(local [(define (name-used-once x_1 ... x_n) exp)]
   name-used-once)
```

can also be written

```
(lambda (x_1 ... x_n) 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 use **`lambda`** to replace

```
(define (eat-apples lst)
  (filter (local [(define (not-symbol-apple? item)
                    (not (symbol=? item 'apple)))]
            not-symbol-apple?)
          lst))
```

with the following:

```
(define (eat-apples lst)
  (filter (lambda (item) (not (symbol=? item 'apple))) lst))
```

But how does this work? As usual, we'll approach it with a trace.

```
(define (eat-apples lst)
  (my-filter (lambda (item) (not (symbol=? item 'apple))) lst))

(eat-apples (list 'pear 'apple))
⇒ (my-filter (lambda (item) (not (symbol=? item 'apple)))) (list 'pear 'apple))
⇒ (cond [(empty? (list 'pear 'apple)) empty]
        [((lambda (item) (not (symbol=?  item 'apple)))
          (first (list 'pear 'apple)))
          (cons (first (list 'pear 'apple))
                (my-filter (lambda (item) (not (symbol=? item 'apple))) (rest
                    (list 'pear 'apple))))]
        [else (my-filter (lambda (item) (not (symbol=? item 'apple)))
                         (rest (list 'pear 'apple)))])
```

What does the underlined expression mean?

```
((lambda (item) (not (symbol=? item 'apple))))  (first (list 'pear 'apple)))
```

The double parentheses indicates that we need to compute the function (the inner expression) to apply to the arguments (the outer expression). In this case, “compute” means create the function using **`lambda`**.

Lambda expressions are already in the simplest form, so the next step in the trace is to reduce the arguments to values:

```
⇒ ((lambda (item) (not (symbol=? item 'apple))))  'pear)
```

Finally, each argument is matched with the corresponding parameter and then substituted into the function’s body expression each place that parameter appears. The entire expression is replaced with the rewritten body expression.

```
⇒ (not (symbol=? 'pear 'apple))
```

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` is available in Intermediate Student with Lambda.

Lambda is the name of the Greek letter λ, which was used as notation in the first formal model of computation.

We'll learn more about its central importance in the history of computation in the last lecture module.

When we first encountered `((make-adder 3) 4)`, we noted the differences in function application:

Before Module 15	**Module 15 and later**
First position in an application must be a built-in or user-defined function.	First position can be an expression (computing the function to be applied). Evaluate it along with the other arguments.
A function name had to follow an open parenthesis.	A function application can have two or more open parentheses in a row: `((make-adder 3) 4)` or `((lambda (x y) (+ x y x)) 1 2)`.

These observations are also true of using **`lambda`**.

We need a rule for evaluating applications where the function being applied is anonymous (a **lambda** expression).

`((lambda (x_1 ... x_n) exp) v_1 ... v_n)` ⇒ `exp'`

where `exp'` is `exp` with all occurrences of `x_1` replaced by `v_1`, all occurrences of `x_2` replaced by `v_2`, and so on.

As an example:

```
((lambda (x y) (* (+ y 4) x)) 5 6)
⇒ (* (+ 6 4) 5)
⇒ ... ⇒ 50
```

> Example: Tracing with lambda (1)

```
(define foo (lambda (x) (+ 10 x)))

(foo 5) ⇒
((lambda (x) (+ 10 x)) 5) ⇒
(+ 10 5) ⇒
15
```

lambda underlies the definition of functions.

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

```
;; a definition of a numerical constant
(define interest-rate 3/100)
;; a definition of a function to compute interest
(define (interest-earned amount) (* interest-rate amount))
```

But there is really only one kind of **define**, which binds a name to a value.

Internally,

```
(define interest-rate 0.03)
(define (interest-earned amount) (* interest-rate amount))
```

is translated to

```
(define interest-earned (lambda (amount) (* interest-rate amount)))
```

which binds the name `interest-earned` to the function value `(lambda (amount) (* interest-rate amount))`.

Here's `make-adder` rewritten using **`lambda`**.

```
(define make-adder
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

What is `((make-adder 3) 4)`?

```
(define make-adder
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

```
(define make-adder (lambda (x) (lambda (y) (+ x y))))
((make-adder 3) 4) ⇒    ;; substitute the lambda 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. Like any other constant, `make-adder` is replaced by its value (the **`lambda`** expression).

Ex. 1

Using **lambda** and `filter` but no named helper functions, write a function that consumes a `(listof Str)` and produces a list containing all the strings that have a length of 4.

```
(keep4 (list "There's"
             "no"
             "fate"
             "but"
             "what"
             "we"
             "make"
             "for"
             "ourselves"))
=> (list "fate" "what" "make")
```

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 substitution step.

Goals of this module

- You should be able to write functions using `lambda` (both consuming and producing).
- You should understand how `lambda` underlies our usual definition of functions.
- You should be able to trace function applications involving `lambda`.

Summary: built-in functions

The following functions and special forms have been introduced in this module:

lambda

You should complete all exercises and assignments using only these and the functions and special forms introduced in earlier modules. The complete list is:

```
* + - ... / < <= = > >= abs add1 and append boolean? ceiling char-alphabetic?
    char-downcase char-lower-case? char-numeric? char-upcase char-upper-case?
    char-whitespace? char<=? char<? char=? char>=? char>? char? check-error check-expect
    check-within cond cons cons? cos define define-struct define/trace e eighth else
    empty? equal? error even? exp expt fifth filter first floor fourth integer? lambda
    length list list->string list? local log max min modulo negative? not number->string
    number? odd? or pi positive? quotient remainder rest reverse round second seventh sgn
    sin sixth sqr sqrt string->list string-append string-downcase string-length
    string-lower-case? string-numeric? string-upcase string-upper-case? string<=? string<?
    string=? string>=? string>? string? sub1 substring symbol=? symbol? tan third zero?
```