λ

CS135 Lecture 18

Finally, we have reached λ ("`lambda`").

While it took a long time to arrive here, λ has always been our destination.

Racket inherits **`lambda`** from earlier programming languages in the same family (Scheme and Lisp) with its roots in a formal, mathematical model for computation called the "λ calculus". Lisp, created by John McCarthy in the late 1950s, adopted many of its foundational ideas from the λ calculus, including its treatment of functions as first-class values and the use of the keyword **`lambda`** to create anonymous functions.

Most modern programming languages provide support for anonymous functions, with some (e.g., Python) also using the keyword **`lambda`**.

L18.0 Anonymous functions

Anonymous functions

In lecture L18 we saw that functions can produce functions.

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

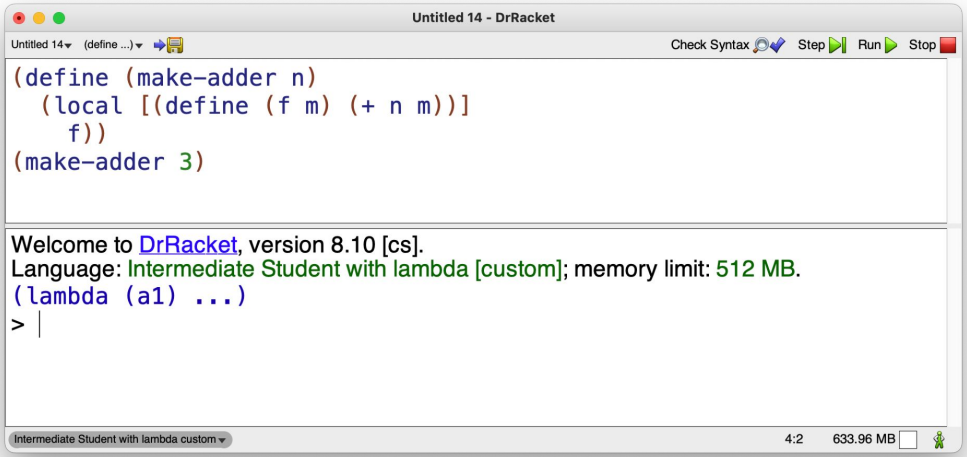
What is the result of the expression below? It's a function, but it has no name.

```
(make-adder 3)
```

Even though it has no name, we can apply it like a defined function:

```
((make-adder 3) 100)⇒ 103
```

Producing anonymous functions



`(lambda (a1) ...)` indicates that the result of `(make-adder 3)` is a function that consumes a single parameter. The function has no name.

`filter` (from lecture L18)

```
;; filter a list to remove all the 'apple
;; eat-apples: (listof Sym) -> (listof Sym)
(define (eat-apples lst)
  (local
    [(define (not-apple sym) (not (symbol=? 'apple sym)))]
    (filter not-apple lst)))

;; filter a list of natural numbers to keep the odd ones
;; keep-odds: (listof Nat) -> (listof Nat)
(define (keep-odds lst)
  (filter odd? lst))
```

Filter with `lambda`

Compared with **`keep-odds`**, the function **`eat-apples`** looks complex.

In particular we defined a local name for **`not-apples`**, which is only used once:

```
(define (not-apple sym) (not (symbol=? 'apple sym)))
```

Instead, we could define a predicate anonymously with lambda:

```
(lambda (sym) (not (symbol=? 'apple sym)))
```

and apply **`filter`** with the predicate:

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

Filter with **`lambda`**

```
;; filter a list to remove all the 'apple
;; eat-apples: (listof Sym) -> (listof Sym)
(define (eat-apples lst)
  (filter (lambda (sym) (not (symbol=? 'apple sym))) lst))

(check-expect
 (eat-apples '(apple eggs bread apple milk bread))
 '(eggs bread milk bread))
```

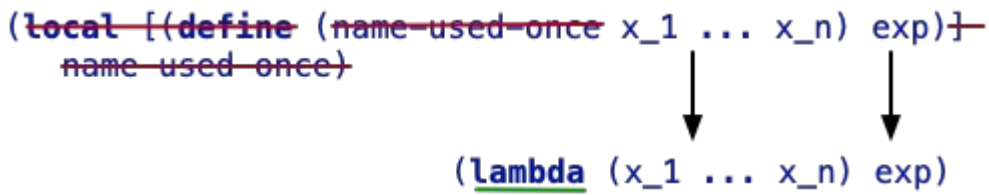
Anonymous functions with **`lambda`**

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

can also be written as a **`lambda`** expression:

```
(lambda (x_1 ... x_n) exp)
```

`lambda` 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`**



Examples

Here are some other examples of using **`lambda`** with **`filter`**:

```
(define lst (list 3 5 9 5 5 4))

(filter (lambda (x) (= 5 x)) lst) ⇒ (list 5 5 5)

(filter (lambda (x) (and (<= 3 x) (<= x 5))) lst)
   ⇒ (list 3 5 5 5 4)

(filter (lambda (s) (char>? s #\Z)) (list #\B #\a #\y))
   ⇒ (list #\a #\y)
```

A source of endless assignment and exam questions.

L18.1 β-reduction

Mathematical context

The “β-reduction” operation, along with a second operation called “α-conversion” (from lecture L16) are the two operations used in the mathematical formulation of computation called “lambda calculus” (or “λ-calculus”). In λ-calculus, β-reduction is the process of applying a function to an argument, where you substitute the argument into the function's body.

In Racket, functions are created using the **`lambda`** form, which directly mirrors the λ-abstraction from λ-calculus. When you call a Racket function defined with **`lambda`**, Racket performs β-reduction by replacing the parameters in the function body with the arguments, effectively “reducing” the expression to its result.

```
((lambda (x y) (+ x y)) 3 4) => (+ 3 4) => 7
```

Substitution rule for **`lambda`**

The substitution rule for applying a lambda expression (β-reduction) is:

`((lambda (`x_1 ... x_n**`)`** e**`)`** v_1 ... v_n**`)`** $\Rightarrow$ e'

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

A lambda expression is already in simplest form. Like the application of a defined function, any simplifications of the expression e are done *after* the parameters are substituted:

```
((lambda (x) (+ x (+ 1 2))) 3) ⇒ (+ 3 (+ 1 2)) ⇒ (+ 3 3) ⇒ 6
```

Making adders

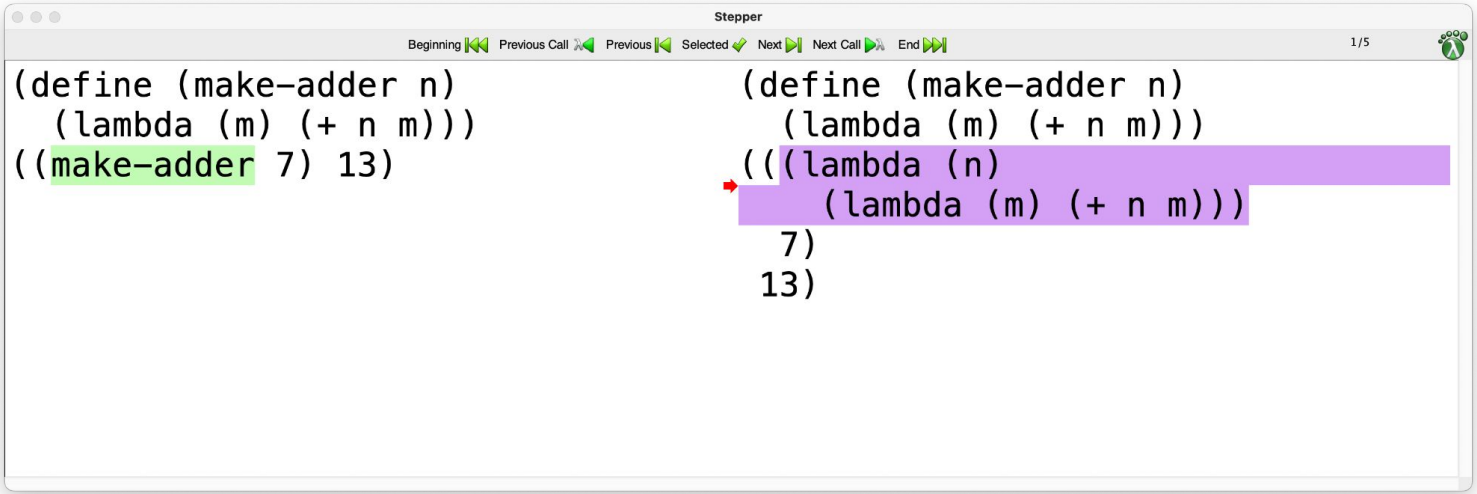
We saw in lecture L17 how **`local`** can be used to produce a function as a value.

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

Now we can use lambda to further **`simplify`** it, avoiding the **`local`** definition.

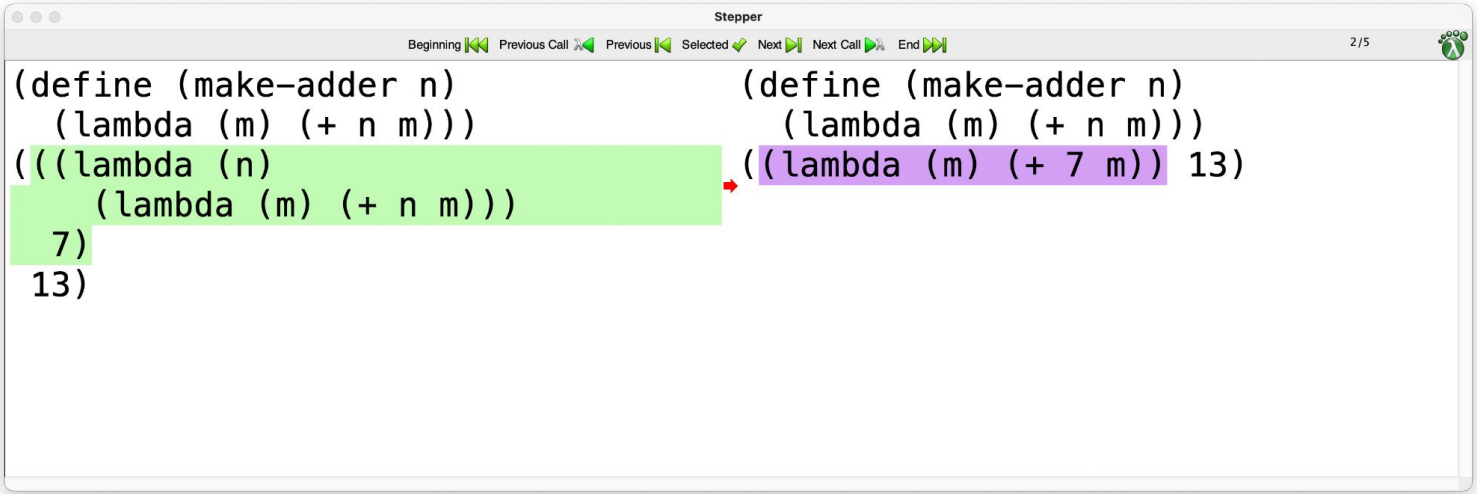
```
(define (make-adder n)
  (lambda (m) (+ n m)))
```

A simple example in the stepper



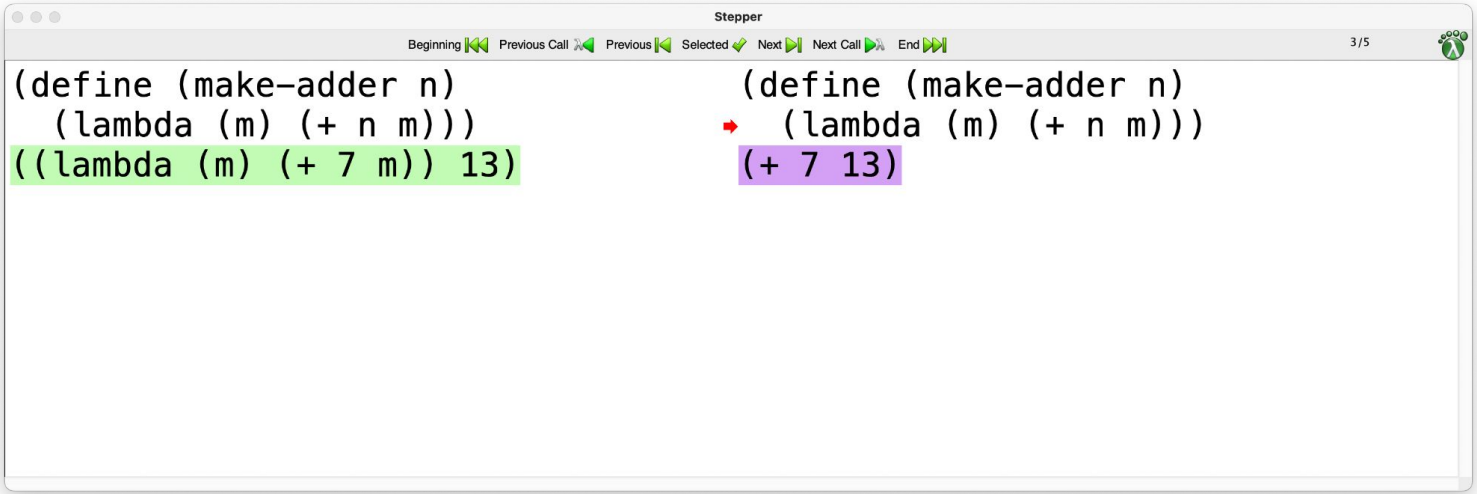
The outer **`lambda`** is from the **`define`**; the inner is the body of **`make-adder`**.

A simple example in the stepper



Application of the outer **`lambda`**.

A simple example in the stepper



Application of the inner **`lambda`**.

L18.2 Stable sorting

Sorting on multiple fields

Here's a list of CS135 students from a different term (not this term).

```
(define cs135
  '(("Banana" "Bob") ("Avocado" "Alice")
    ("Banana" "Carol") ("Aardvark" "Bob")
    ("Avocado" "Carol") ("Aardvark" "Alice")
    ("Cucumber" "Alice") ("Avocado" "Bob")
    ("Cucumber" "Carol") ("Bear" "Bob")))
```

Each list element is a `("`*family name*`" "`*given name*`")` pair. As you can see, while no two people have the same name, nobody has a unique family or given name,

Sorting on multiple fields

We want to sort this list according to English-language conventions: alphabetically, i.e., lexicographically, by family name and then by given name.

```
(define sorted-cs135
  '(("Aardvark" "Alice") ("Aardvark" "Bob")
    ("Avocado" "Alice") ("Avocado" "Bob")
    ("Avocado" "Carol") ("Banana" "Bob")
    ("Banana" "Carol") ("Bear" "Bob")
    ("Cucumber" "Alice") ("Cucumber" "Carol")))
```

For simplicity, we compare names with **`string<?`**, but be warned that sorting names with **`string<?`**, i.e., by codepoints, is not universally correct.

Reminder: Merging two sorted lists (from lecture L10).

```
;; merge: (listof Num) (listof Num) -> (listof Num)
;; Requires: lists must be sorted in increasing order
(define (merge lst1 lst2)
  (cond [(empty? lst1) lst2]
        [(empty? lst2) lst1]
        [(< (first lst1) (first lst2))
         (cons (first lst1) (merge (rest lst1) lst2))]
        [(> (first lst1) (first lst2))
         (cons (first lst2) (merge lst1 (rest lst2)))]
        [else (cons (first lst1)
                    (cons (first lst2)
                          (merge (rest lst1) (rest lst2))))]))
```

We can simplify **`merge`**

```
;; merge: (listof Num) (listof Num) -> (listof Num)
;; Requires: lists must be sorted in increasing order
(define (merge lst1 lst2)
  (cond [(empty? lst1) lst2]
        [(empty? lst2) lst1]
        [(< (first lst1) (first lst2))
         (cons (first lst1) (merge (rest lst1) lst2))]
        [else
         (cons (first lst2) (merge lst1 (rest lst2)))]))
```

Notice that the function still works for equal elements, preferring the second list.

Generalizing `merge`

What about if we want to sort strings or numbers in decreasing order?

Without first-class functions we need to rewrite **`merge`** to replace the **<** and **>**, but now we can use write merge once and pass pass predicates for the comparisons.

```
(check-expect
 (merge string<? (list "Alice" "Carol") (list "Bob"))
 (list "Alice" "Bob" "Carol"))

(check-expect (merge > '(6 4 2) '(7 5 3 1))
              '(7 6 5 4 3 2 1))
```

Generalizing **`merge`**

```
(define (merge <? lst1 lst2)
  (cond [(empty? lst1) lst2]
        [(empty? lst2) lst1]
        [(<? (first lst1) (first lst2))
         (cons (first lst1) (merge <? (rest lst1) lst2))]
        [else
         (cons (first lst2) (merge <? lst1 (rest lst2)))]))
(check-expect
 (merge string<? (list "Alice" "Carol") (list "Bob"))
 (list "Alice" "Bob" "Carol"))
(check-expect (merge > '(6 4 2) '(7 5 3 1))
              '(7 6 5 4 3 2 1))
```

Stable sorting

A “stable” sorting algorithm is one that preserves the relative order of elements that are considered equal, i.e. `(<? x y)` and `(<? y x)` are both `false`.

For example, if two people in our list of names have the same family name, a stable sort on the given name guarantees that they appear in the same order in the sorted list.

```
(define given-sorted-cs135
  '(("Avocado" "Alice") ("Aardvark" "Alice")
    ("Cucumber" "Alice") ("Banana" "Bob")
    ("Aardvark" "Bob") ("Avocado" "Bob")
    ("Bear" "Bob") ("Banana" "Carol")
    ("Avocado" "Carol") ("Cucumber" "Carol")))
```

Stable sorting

```
'(("Banana" "Bob") ("Avocado" "Alice")
   ("Banana" "Carol") ("Aardvark" "Bob")
   ("Avocado" "Carol") ("Aardvark" "Alice")
   ("Cucumber" "Alice") ("Avocado" "Bob")
   ("Cucumber" "Carol") ("Bear" "Bob")))

 '(("Avocado" "Alice") ("Aardvark" "Alice")
   ("Cucumber" "Alice") ("Banana" "Bob")
   ("Aardvark" "Bob") ("Avocado" "Bob")
   ("Bear" "Bob") ("Banana" "Carol")
   ("Avocado" "Carol") ("Cucumber" "Carol"))
```

Stable sort on given name.

Stable `mergesort`

For `mergesort` to be stable, it must preserve the relative order of equal elements when we split the lists:

1. Split the list into two lists of nearly equal length: first half and second half.

   `(list 8 4 3 9 1 6 2 5 0 7)` ⇒ `(list 8 4 3 9 1) (list 6 2 5 0 7)`

2. Stably sort each of the two shorter lists.

   `(list 1 3 4 8 9) (list 0 2 5 6 7)`

3. Merge the sorted lists. When elements are equal choose from the first half.

   `(list 0 1 2 3 4 5 6 7 8 9)`

Splitting a list into a first half and second half

```
(define (first-n n lst)
  (cond [(or (empty? lst) (zero? n)) empty]
        [else (cons (first lst)
                    (first-n (sub1 n) (rest lst)))]))
(check-expect (first-n 3 '(1 2 3 4 5 6 7))'(1 2 3))

(define (chop-n n lst)
  (cond [(or (empty? lst) (zero? n)) lst]
        [else (chop-n (sub1 n) (rest lst))]))
(check-expect (chop-n 3 '(1 2 3 4 5 6 7)) '(4 5 6 7))
```

Splitting a list into a first half and second half

```
;; split a list into nearly equal halves
;; split: (listof Any) -> (listof Any) (listof Any)
(define (split lst)
  (local
    [(define n (quotient (length lst) 2))]
    (list (first-n n lst) (chop-n n lst))))

(check-expect
 (split '(1 2 3 4 5 6 7)) '((1 2 3) (4 5 6 7)))
(check-expect
 (split '(a b c d e f)) '((a b c) (d e f))
```

A function that makes stable sorting functions

```
;; produce a sort function from a predicate
;; make-sort: (X X -> Bool) -> ((listof X) -> (listof X))
(define (make-sort <?)
  (local
    [(define (mergesort lst)
       (cond [(or (empty? lst) (empty? (rest lst))) lst]
             [else
              (local
                [(define s (split lst))]
                (merge <?
                       (mergesort (second s))
                       (mergesort (first s))))]))]
    mergesort))
```

Stable sorting by given names

```
(define stable-sort-given-names
  (make-sort
   (lambda (x y) (string<? (second x) (second y)))))

(define given-sorted-cs135
  '(("Avocado" "Alice") ("Aardvark" "Alice")
    ("Cucumber" "Alice") ("Banana" "Bob")
    ("Aardvark" "Bob") ("Avocado" "Bob")
    ("Bear" "Bob") ("Banana" "Carol")
    ("Avocado" "Carol") ("Cucumber" "Carol")))
(check-expect
 (stable-sort-given-names cs135) given-sorted-cs135)
```

Composing sorting functions

Because **`make-sort`** makes stable sorting functions we can compose them to sort on multiple fields.

```
(define (sort-names lst)
 ((make-sort
   (lambda (x y) (string<? (first x) (first y))))
  ((make-sort
    (lambda (x y) (string<? (second x) (second y))))
   lst)))
(check-expect (sort-names cs135) sorted-cs135)
```

The sorter for given names is applied before the sorter for family names. We say that we are sorting with family names as the “primary key” and given names as the “secondary key”.

Sorting should be stable

Stable sorting ensures that the relative order of equal elements remains unchanged after a sort. This property is essential in multi-step sorting processes.

Spreadsheets, (e.g., Excel, Google Sheets) sort stably.

Imagine you have a spreadsheet with employee records. One column contains employee names and another column contains their departments. Initially, the data is sorted by name. If you then sort by department using a stable sorting algorithm, employees within the same department will remain in alphabetical order by name. This means you maintain employee name as a secondary sort key.

When writing a sorting function, care must be taken to ensure it is stable.

Racket's built-in `sort` is stable

Racket has a built-in `sort` that you can now use.

```
sort: (listof X) (X -> Bool) -> (listof X)
```

The list is the first argument and the predicate is the second argument.

```
(check-expect
 (sort '(8 4 3 9 1 6 2 5 0 7) >)
 '(9 8 7 6 5 4 3 2 1 0))
(check-expect
 (sort '("Bob" "Carol" "Alice") string<?)
 '("Alice" "Bob" "Carol"))
```

Sorting in Python is stable, but the default sort in C or C++ may not be.

L18 Summary

L18: You should know

- How to create anonymous functions with **`lambda`**.
- How to apply anonymous functions.
- The substitution rule for lambda ("β-reduction").
- The importance of stable sorting.
- How to implement stable sorting in mergesort.
- How to use Racket's built-in **`sort`** function.

L18: Allowed constructs

Newly allowed constructs:

```
lambda sort
```

Previously allowed constructs:

```
( ) [ ] + - * / = < > <= >= ; '
abs acos add1 and append asin atan char? char=? char<?
check-expect check-within cond cons cons? cos define e else
empty empty? even? exp expt false filter first inexact?
integer? length list list? local log max min not number?
odd? or pi quotient rational? remainder rest reverse second
sin sqr sqrt string? string=? string<? string->list
list->string sub1 symbol? symbol=? tan third true zero?
listof Any anyof Atom Bool Char Int Nat Num Rat Str Sym
```