λ

CS135 Lecture 17

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.

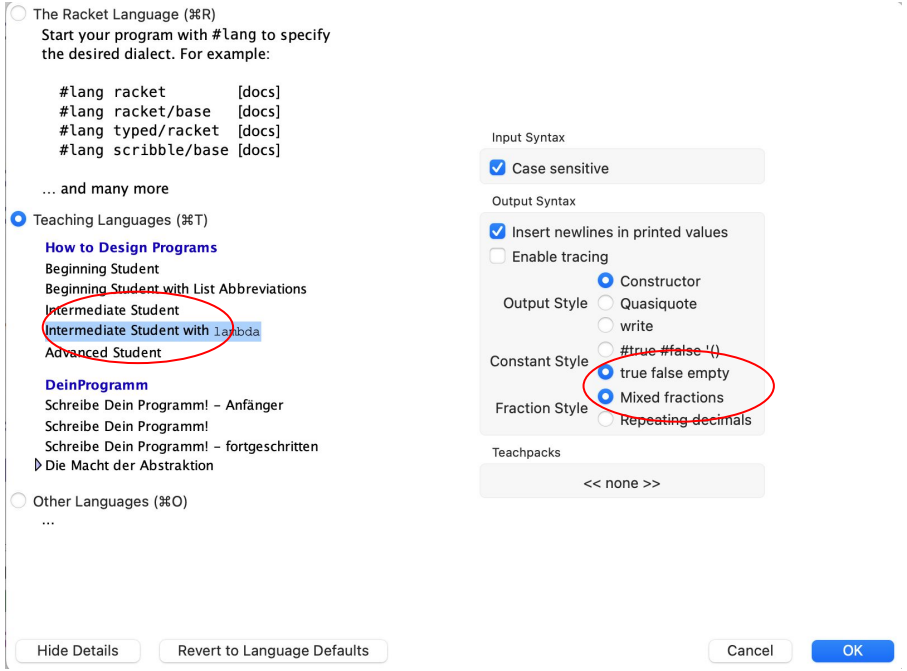
Most modern programming languages provide support for anonymous functions, with some (e.g., Python) also using the keyword **`lambda`**.

Intermediate Student with `lambda`

It's time to change language level yet again, this time to "Intermediate Student with **`lambda`**".

This is our last switch. We never reach "Advanced Student".

Don't forget to set "Constant Style" and "Fraction Style" as shown on the right.



L17.0 Anonymous functions

Anonymous functions with **`lambda`**

A **`lambda`** expression allows us to create an anonymous function, i.e., a function that is not bound to any name. An anonymous function can be written as a ``**`lambda`** expression'' with the form:

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

A **`lambda`** expression is a value in the same sense that **`12`**, **`(cons 1 empty)`**, and **`true`** are values, for example:

`(lambda (x) (+ 10 x))`
is a function that adds 10 to its argument.

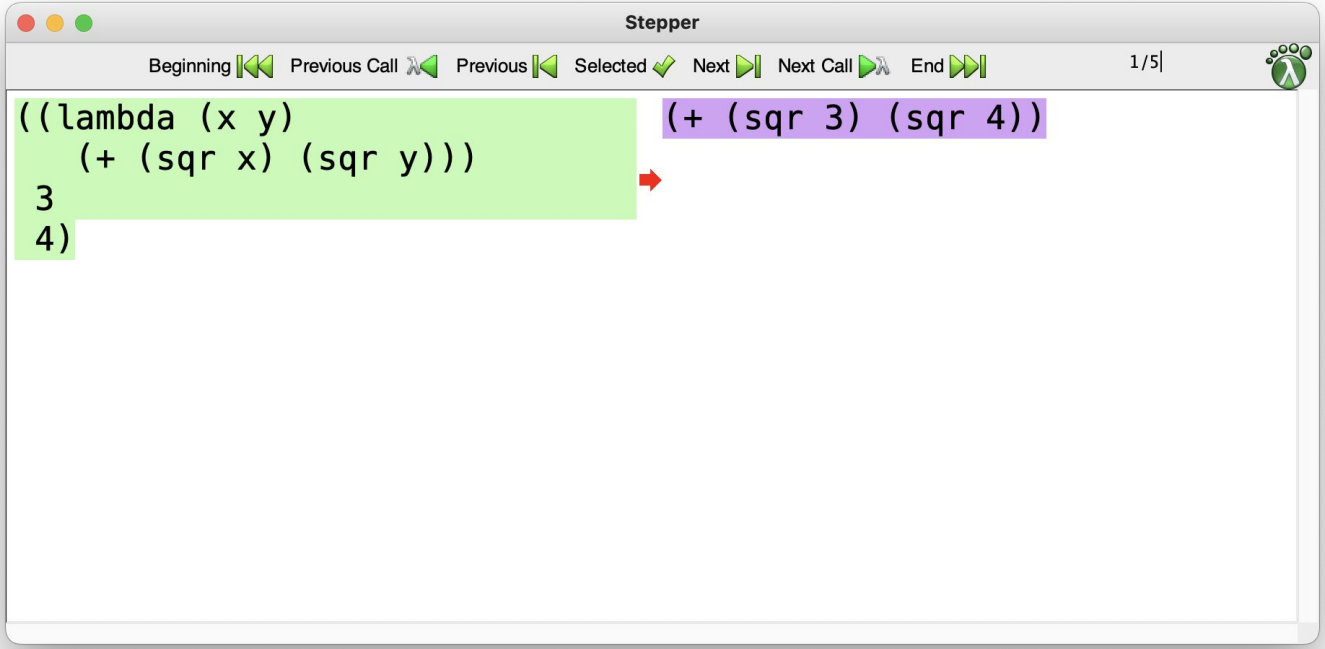
`(lambda (x y) (+ (sqr x) (sqr y)))`
is a function that computes the sum of squares of its arguments.

Function application

A **`lambda`** expression is a function. It can be applied to arguments in the same way as any other function.

```
((lambda (x) (+ 10 x)) 7) ⇒ (+ 10 7) ⇒ 17

((lambda (x y) (+ (sqr x) (sqr y))) 3 4)
⇒ (+ (sqr 3) (sqr 4))
⇒ (+ 9 16)
⇒ 25
```

Function application in the stepper



Function definitions

In “Intermediate student with **`lambda`**” function definitions are first converted into lambda expressions before they are applied.

```
(define (g x y) (+ (sqr x) (sqr y)))
(g -3 4)
```

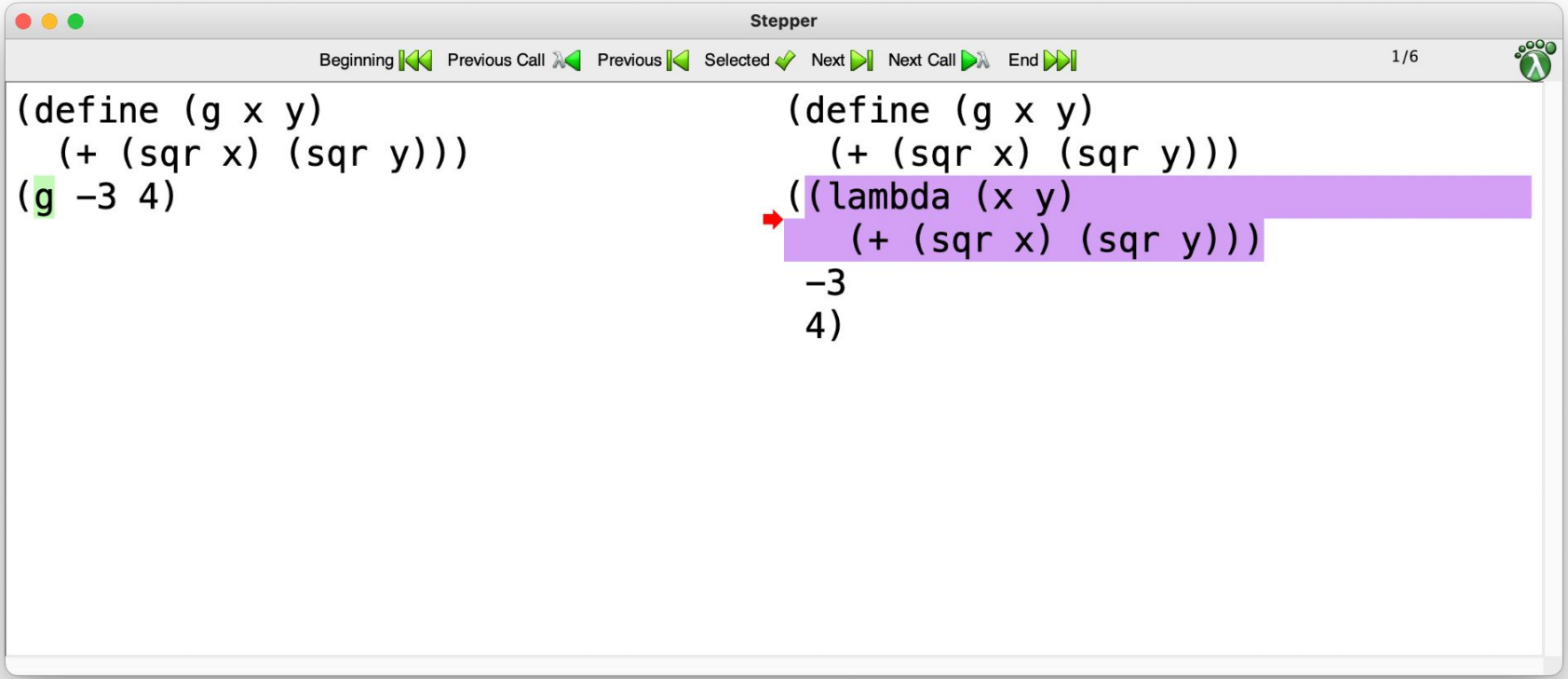
⇒

```
(define (g x y) (+ (sqr x) (sqr y)))
((lambda (x y) (+ (sqr x) (sqr y))) -3 4)
```

⇒

```
(define (g x y) (+ (sqr x) (sqr y)))
(+ (sqr -3) (sqr 4))⇒(+ 9 (sqr 4))⇒(+ 9 16)⇒25
```

Function application



Two ways to define a function

These definitions are equivalent:

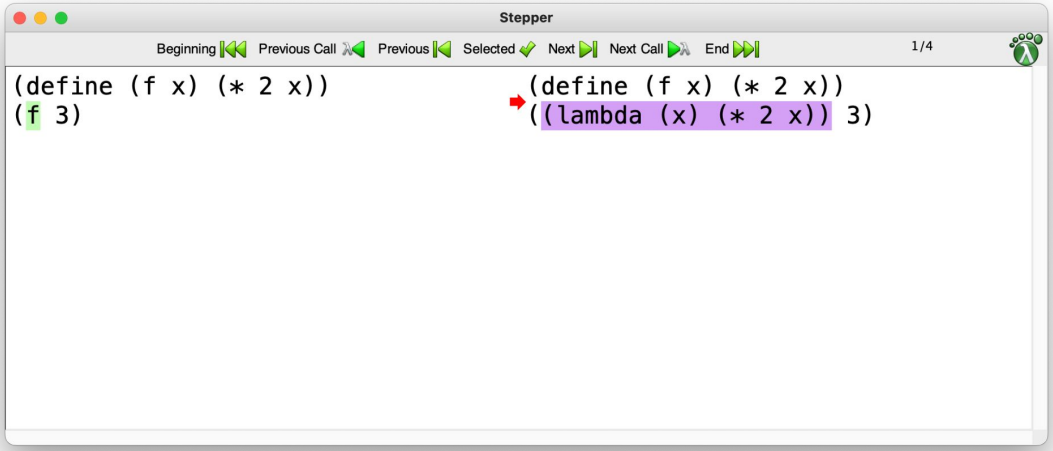
```
(define (g x y) (+ (sqr x) (sqr y)))

(define g (lambda (x y) (+ (sqr x) (sqr y))))
```

We no longer need a **`define`** for constants and a **`define`** for functions.

In “Intermediate student with **`lambda`**” the application of a user-defined function requires two steps”:

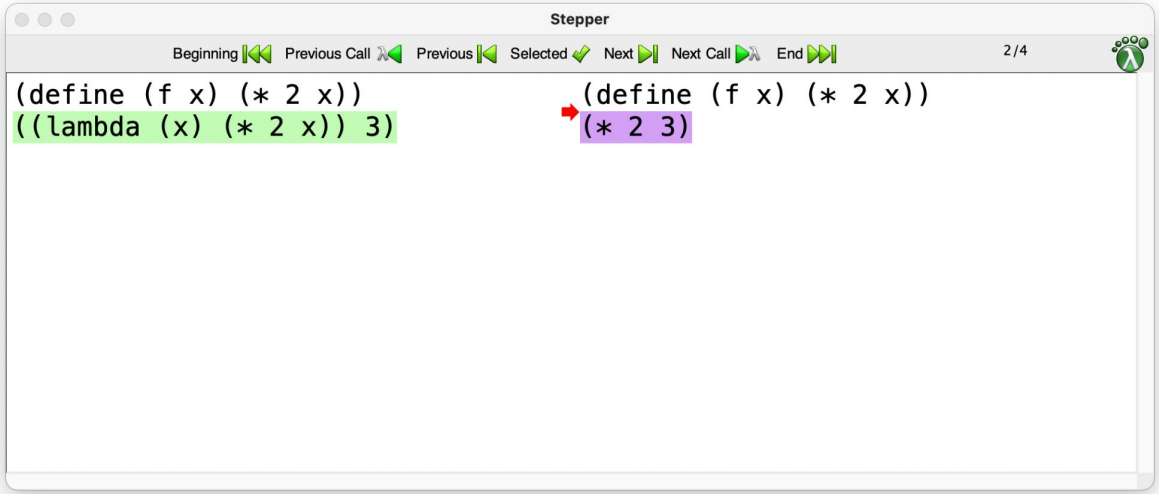
1. The name of the function is replaced by the corresponding **`lambda`** expression.
2. The parameters are substituted into the **`lambda`** expression.

1. Replace the name of the function with the `lambda`



This step is not new. Now function definitions are handled like constant definitions, where the value of the constant (the `lambda`) replaces the name of the function.

2. Substitute parameters into the `lambda` expression



Substituting parameters into a lambda expression is a new kind of substitution step, called "β-reduction".

L17.1 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. Any simplifications of the expression *e* are done *after* the parameters are substituted:

`((lambda (x) (+ x (+ 1 2))) 3)` $\Rightarrow$ **`(+ 3 (+ 1 2))`** $\Rightarrow$ **`(+ 3 3)`** $\Rightarrow$ **`6`**

L17.2 Producing functions

Functions as first-class values

In Racket (at the “Intermediate student with `lambda`” level) functions themselves are “first-class” values.

They can be produced, consumed, and appear in lists like other values, such as numbers, symbols and strings. While functions aren’t usually considered “atoms”, you can think of them that way.

In remainder of this lecture, we’ll consider functions that produce and consume other functions.

In the next lecture, we’ll consider more complex examples, including lists of functions and composition.

Making adders

The built-in function **`add1`**, adds a fixed quantity (one) to any number.

We could, of course, build our own versions to add other quantities.

```
(define (add10 x) (+ 10 x))
(check-expect (add10 10) 20)

(define (add-half x) (+ 1/2 x))
(check-expect (add-half 1) 3/2)
```

If functions are first-class values, we can write a function that produces an adder for any quantity.

Making adders

We saw in lecture L16 how `local` could be used to define new functions to be used in evaluating the body of the `local`.

But now, because functions are values, the body of the `local` can produce such a function as a value.

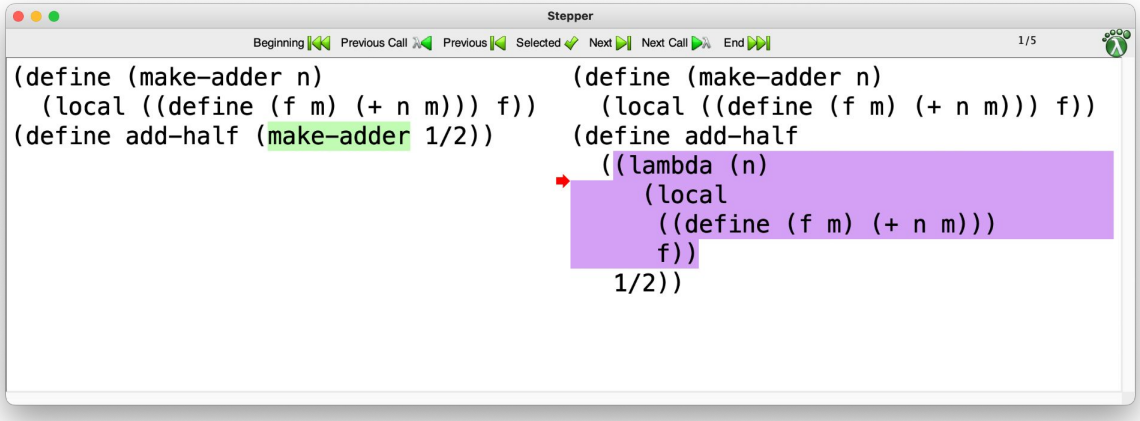
```
(define (make-adder n)
  (local
    [(define (f m) (+ n m))]
    f))
```

Notice how n is somehow captured inside the produced function.

Making adders

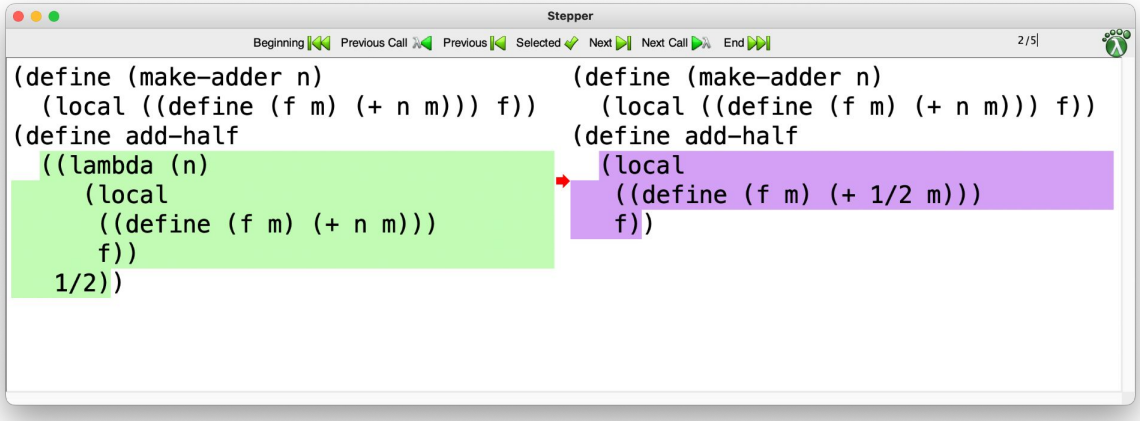
```
;; make-adder: Num -> (Num -> Num)
(define (make-adder n)
  (local
    [(define (f m) (+ n m))]
    f))

(define add10 (make-adder 10))
(check-expect (add10 10) 20)

(define add-half (make-adder 1/2))
(check-expect (add-half 1) 3/2)
```

Making an adder



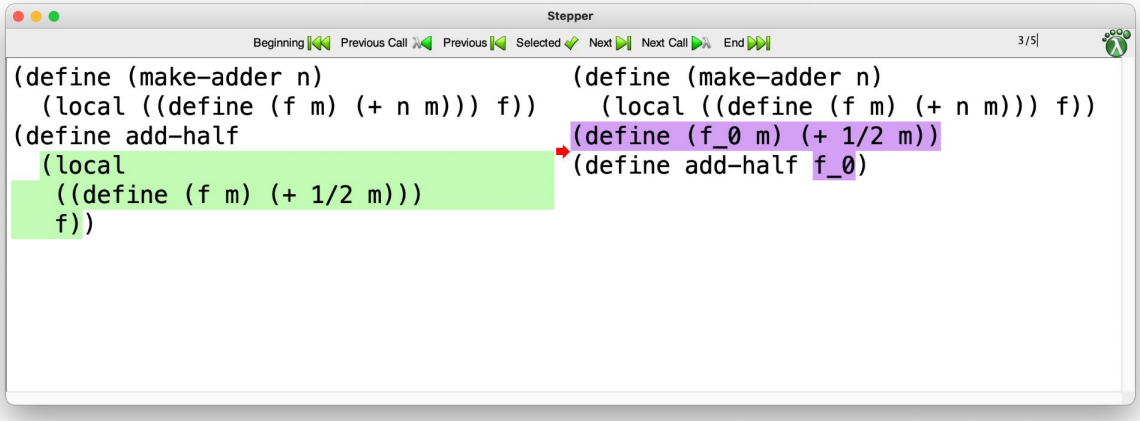
Replace **`make-addr`** with its **`lambda`** value.

Making an adder



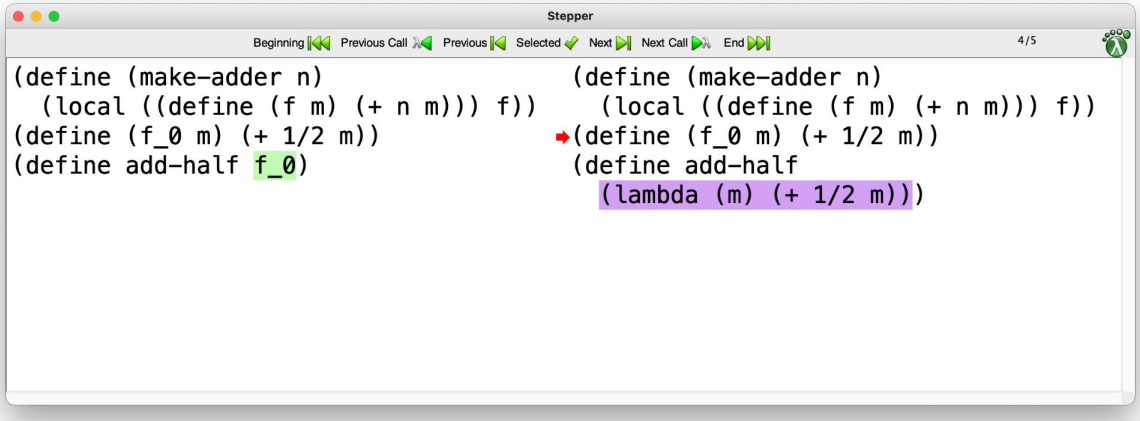
The value of `n` is captured by the `local` through β-reduction.

Making an adder



α-conversion assigns a fresh identifier and lifts the definition to the top level.

Making an adder



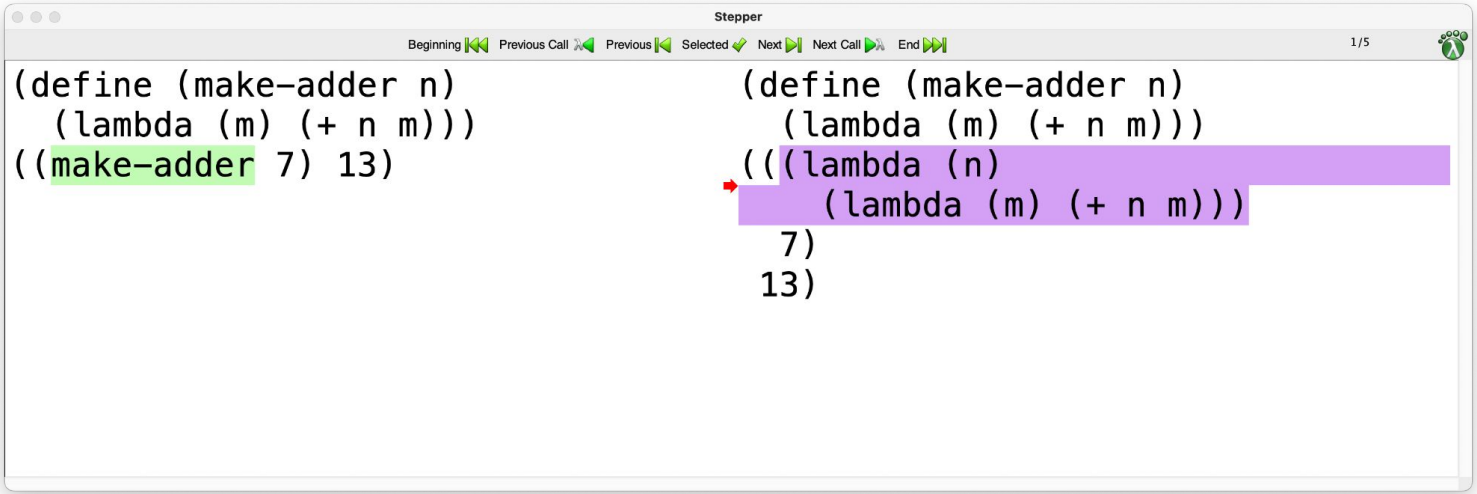
The **`add-half`** function is defined as a **`lambda`** value.

Making adders with `lambda`

We can also use `lambda` to avoid the `local` definition.

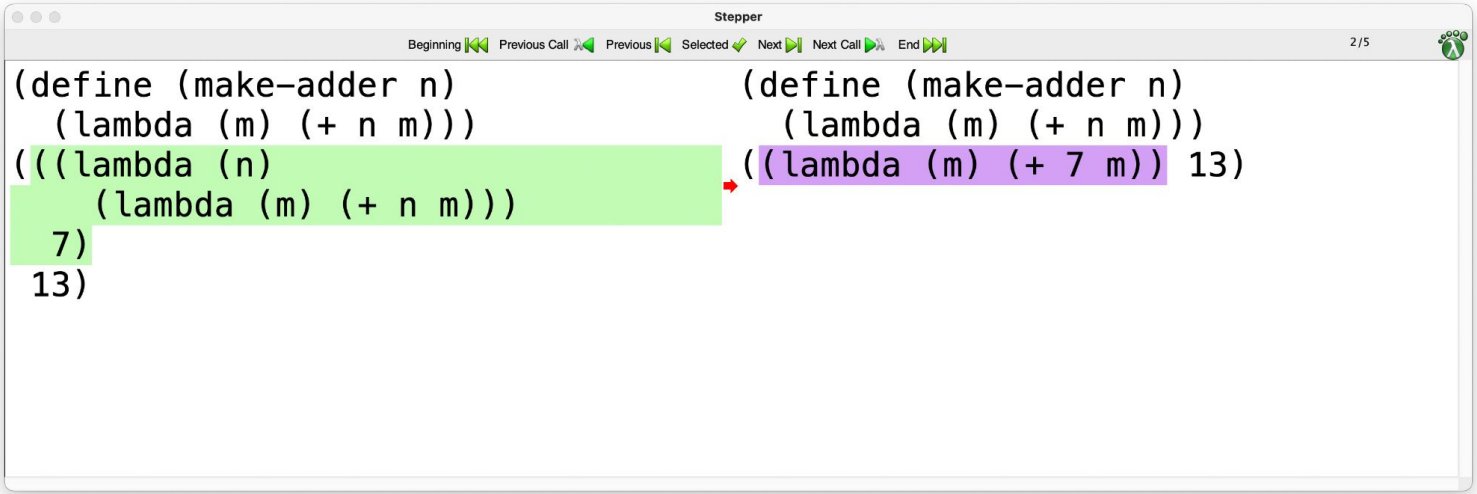
```
;; make-adder: Num -> (Num -> Num)
(define (make-adder n)
  (lambda (m) (+ n m)))

(define add10 (make-adder 10))
(check-expect (add10 10) 20)

(define add-half (make-adder 1/2))
(check-expect (add-half 1) 3/2)
```

Making an adder with **`lambda`**



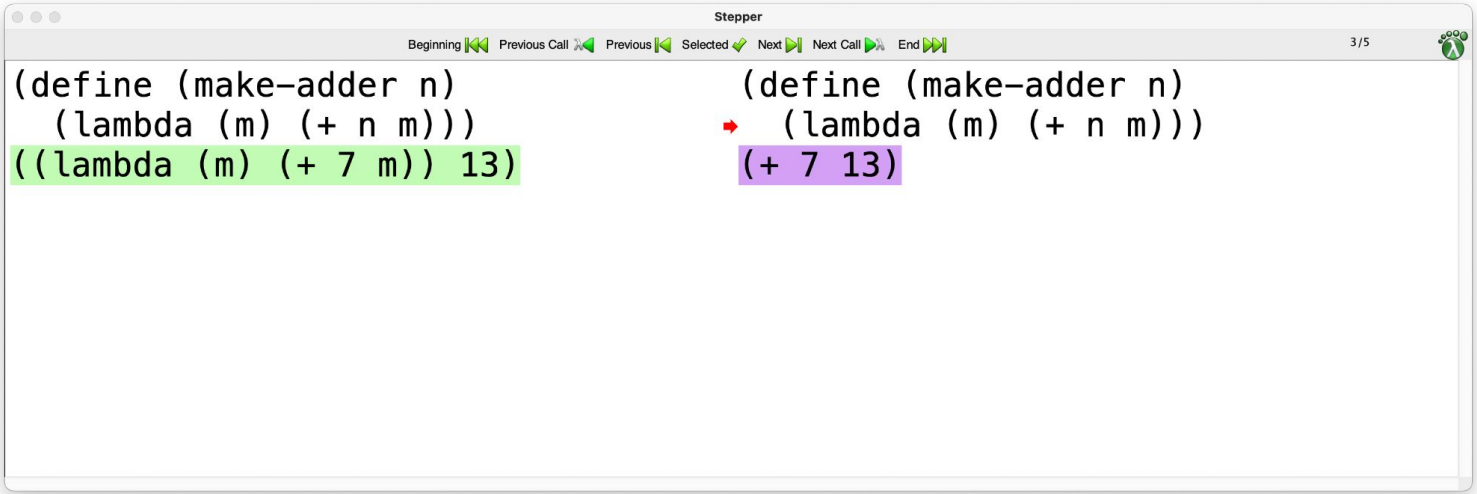
The outer **`lambda`** is from the **`define`**; the inner is the body of **`make-adder`**.

Making an adder with `lambda`



Application of the outer `lambda`.

Making an adder with `lambda`



Application of the inner `lambda`.

L17.3 Consuming functions

Eating apples from a list of symbols (lecture L07)

```
(define (eat-apples lst)
  (cond [(empty? lst) empty]
        [(symbol=? 'apple (first lst))
         (eat-apples (rest lst))]
        [else (cons (first lst) (eat-apples (rest lst)))]))

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

Keeping odd natural numbers in a list of numbers

```
(define (keep-odds lst)
  (cond [(empty? lst) empty]
        [(not (odd? (first lst)))
         (keep-odds (rest lst))]
        [else (cons (first lst) (keep-odds (rest lst)))]))

(check-expect
 (keep-odds '(1 1 3 4 6 5 8 7)) '(1 1 3 5 7))
```

Apart from the name, only the predicate is different.

```
(define (eat-apples lst)
  (cond [(empty? lst) empty]
        [(symbol=? 'apple (first lst))
         (eat-apples (rest lst))]
        [else (cons (first lst) (eat-apples (rest lst)))]))

(define (keep-odds lst)
  (cond [(empty? lst) empty]
        [(not (odd? (first lst)))
         (keep-odds (rest lst))]
        [else (cons (first lst) (keep-odds (rest lst)))]))
```

Keeping anything

```
;; keep elements of a list for which the predicate is true
;; keep: (X -> Bool) (listof X) -> (listof X)
;;
(define (keep pred? lst)
  (cond [(empty? lst) empty]
        [(pred? (first lst))
         (cons (first lst) (keep pred? (rest lst)))]
        [else (keep pred? (rest lst))]))
```

Eating apples and keeping odds

```
(define (eat-apples lst)
  (local
    [(define (not-apple? sym) (not (symbol=? 'apple sym)))]
    (keep not-apple? lst)))
(check-expect
 (eat-apples '(apple eggs bread apple milk bread))
 '(eggs bread milk bread))

(define (keep-odds lst)
  (keep odd? lst))
(check-expect
 (keep-odds '(1 1 3 4 6 5 8 7)) '(1 1 3 5 7))
```

filter

Racket has a built-in function that does what **keep** does, which you can now use.

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

(define (keep-odds lst)
  (filter odd? lst))
```

Like other built-in functions that operation on entire lists, you should think of **filter** as linear in the size of the consumed list.

Filter with `lambda`

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

In particular we defined a local name for **`not-apple?`**, 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 symbols
;; 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))
```

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.

L17.4 Contracts

Higher-order functions

"Higher-order functions" are functions that consume other functions as arguments and/or produce them as results. We have seen multiple examples of higher-order functions in this lecture.

Languages that support functions as first-class values, like Racket, allow us to write higher-order functions, which can make our code simpler and more flexible.

For example, by changing predicates we can `filter` lists in any way we like, without writing a new recursive function.

Unfortunately, writing contracts for higher-order functions introduces some new complexities.

Types, subtypes, and supertypes

Recall from lecture L01 that **`Nat`**, **`Int`** and **`Rat`** are *subtypes* of **`Num`**.

Since any **`Nat`** is also an **`Int`**, we say:

- **`Nat`** is a *subtype* of **`Int`**, and
- **`Int`** is a *supertype* of **`Nat`**.

We can write the relationship between them as: **`Nat`** $\subseteq$ **`Int`** $\subseteq$ **`Rat`** $\subseteq$ **`Num`**

In Racket, types are checked *dynamically*, as the substitution rules are applied.

Our contracts attempt to document typing relationships *statically*, at the time we write a function.

Contracts for higher-order functions

Earlier in the lecture we wrote a contract and purpose for **`keep`** as

```
;; keep elements of a list for which the predicate is true
;; keep: (X -> Bool) (listof X) -> (listof X)
```

A more accurate contract might be:

```
;; keep: (Z -> Bool) (listof Y) -> (listof X)
```

where **`X`** ⊆ **`Y`** ⊆ **`Z`**.

`X`, **`Y`** and **`Z`** are called “type variables”, which are used to show relationships between types.

Contracts for higher-order functions

Suppose we want to filter a list of pets (e.g., `'cat`, `'budgie`, `'goldfish`) to keep only the dogs (e.g., `'hound`, `'beagle`).

Suppose we have the subtype relationship `Dog` ⊆ `Pet` ⊆ `Animal`, in addition we have a predicate `dog?: Animal -> Bool`.

If keep has the contract

```
;; keep: (X -> Bool) (listof X) -> (listof X)
```

then we can't use `dog?` to filter a `(listof Pet)` because `dog?` consumes a `Animal`, not a `Pet`.

However, we "know" that it's okay to use `dog?` because every `Pet` is an `Animal`.

Covariance and contravariance

Covariance means that if you have a subtype relationship between types, that relationship is preserved when those types appear in “output positions”. For example, any function producing a `(listof Dog)` can be used where a function producing a `(listof Animals)` is expected.

Contravariance applies to function “input positions”, i.e, parameters. It reverses the subtyping: a function that consumes an `Animal` can serve as a substitute where a function consuming a `Dog` is expected.

We don’t expect you to know these terms for exams, this information is just for context and extra insight.

Contracts for higher-order functions

In CS135, to avoid the complexity of multiple type variables, covariance, contravariance, we will write contracts for higher-order functions with a single type variable.

```
;; keep: (X -> Bool) (listof X) -> (listof X)
```

We will not ask **you** to write contracts for higher order functions, but you should understand them when you see them on exams or assignments.

If you are interested in exploring types and subtyping in the context of higher-order functions, you might be interested in the Scala programming language, which is widely used in industry. Knowing Racket will be a big help when learning Scala.

https://docs.scala-lang.org

L17 Summary

L17: You should know

- How to create anonymous functions with **`lambda`**.
- How to apply anonymous functions.
- The substitution rule for lambda ("β-reduction").
- How to write functions that produce and consume other functions.
- How to understand contracts for higher-order functions.

You can even write `λ` instead of `lambda` in Racket

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

(define add7 (make-adder 7))

(check-expect (add7 13) 20)
```

But we will generally spell it out as "`lambda`" (and `λ` is not officially on the allowed list).

L17: Allowed constructs

Newly allowed constructs:

```
filter lambda
```

Previously allowed constructs:

```
( ) [ ] + - * / = < > <= >= ; '
abs acos add1 and append asin atan boolean? char? char=?
char<? check-expect check-within cond cons cons? cos define
e else empty empty? exp expt false 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
```