CSC324 Principles of Programming Languages

Lecture 2

September 18, 2019

Last class: Lambda Calculus and Racket

- The Lambda Calculus
- Functional Programming
- Recursion

```
(define (add1-list lst)
  (if (null? lst)
    (list)
    (cons (+ 1 (first lst))
          (add1-list (rest lst)))))
```

Today: Functional Programming

Today's lecture is about *functions* of various kinds:

- **Pattern matching**: a different way of writing functions
- **Tail recursion**: making recursion fast
- **Higher-order functions**: functions that take functions as arguments
- **Currying**: functions that takes in *some* of its parameters

. . . but first, let's talk a bit about Haskell

Haskell Functions

- **Anonymous functions**: `\<param> ... -> <expr>`
  - Example: `\x -> x + 1`
  - Example: `\x y -> x * y + 1`

Haskell Functions

- **Anonymous functions**: `\<param> ... -> <expr>`
  - Example: `\x -> x + 1`
  - Example: `\x y -> x * y + 1`
- **Function application**: `<function> <arg> ...`
  - Example: `(\x -> x + 1) 3`
  - Example: `(\x y -> x * y + 1) 4 5`

Haskell Name Bindings

- **Global Name Binding**: `<id> = <expr>`
  - Example: `z = 2`
  - Example: `f = \x -> x * 2`
  - Alternative: `f x = x * 2`
- **Local Name Binding**:
  - Example: `let x = 1 in x`

Haskell Lists

```
Prelude> [1, 2, 3, 4]
[1,2,3,4]
Prelude> 1:2:3:4:[]
[1,2,3,4]
Prelude> head [1, 2, 3, 4]
1
Prelude> tail [1, 2, 3, 4]
[2,3,4]
```

Recursion in Haskell

Write a version of this Racket function in Haskell

```
(define (add1-list lst)
  (if (null? lst)
    (list)
    (cons (+ 1 (first lst))
          (add1-list (rest lst)))))
```

Recursion in Haskell

Write a version of this Racket function in Haskell

```
(define (add1-list lst)
  (if (null? lst)
    (list)
    (cons (+ 1 (first lst))
          (add1-list (rest lst)))))
```

Haskell:

```
add1_list xs =
    if xs == []
        then []
        else (1 + head xs):(add1_list (tail xs))
```

Multiple If Statements in Haskell

```
num_to_word num =
  if num == 1 then
    "one"
  else if num == 2 then
    "two"
  else if num == 3 then
    "three"
  else
    "error"
```

This function can be written more elegantly using **pattern matching**.

Pattern Matching

Value Pattern-Matching in Haskell

Same code written using pattern-matching on **values**:

```
num_to_word 1 = "one"
num_to_word 2 = "two"
num_to_word 3 = "three"
num_to_word _ = "error"
```

Evaluation:

- One function body per value
- Patterns are checked from top to bottom; the first that matches is used
- "Catch all" pattern at the very end

Value Pattern-Matching for add1_list

Can we use value pattern-matching in this code?

```
add1_list xs =
    if xs == []
        then []
        else (1 + head xs):(add1_list (tail xs))
```

Value Pattern-Matching for add1_list

Can we use value pattern-matching in this code?

```
add1_list xs =
    if xs == []
        then []
        else (1 + head xs):(add1_list (tail xs))
```

Yes!

```
add1_list [] = []
add1_list xs = (1 + head x):(add1_list (tail xs))
```

(We can do even better in a few slides)

Value Pattern-Matching in Racket

We can also use value pattern-matching in Racket using `define/match`:

```
(define/match (f x)
  [(1) "one"]
  [(2) "two"]
  [(3) "three"]
  [(x) "error"])
```

Structural Pattern-matching

Rather than pattern-match on values, we can pattern match on the *structure* of data, and *deconstruct* its elements.

Example:

- A list in Racket is either:
  - An empty list: `null` or `'()`
  - A `cons` of an element to a list: `(cons x lst)`
- Likewise, a list in Haskell is either `[]` or `x:lst`

Structural Pattern-matching in Racket

We can **deconstruct** a list in our second pattern:

```
(define/match (add1-list lst)
  [((list))       '()]
  [((cons x xs)) (cons (+ 1 x)
                       (add1-list xs))])
```

Structural Pattern-matching in Haskell

We can **deconstruct** a list in our second pattern:

```
add1_list []     = []
add1_list (x:xs) = (1 + x):(add1_list xs)
```

Worksheet (Questions 1-2)

Tail Recursion

Recursion in Python is Slow

```
define my_sum(lst):
    if len(lst) == 0:
        return 0
    return lst[0] + my_sum(lst[i:])
```

Initial function call: `sum([1, 2, 3, 4])`

- Need to compute: `1 + sum([2, 3, 4])`
- Initial function call put on hold until we evaluate `sum([2, 3, 4])`

Call Stack

- Every time a function is put on hold, it takes up space on the **call stack**.
- Memory used scales linearly with the number of recursive calls

Call Stack - Python Visualizer

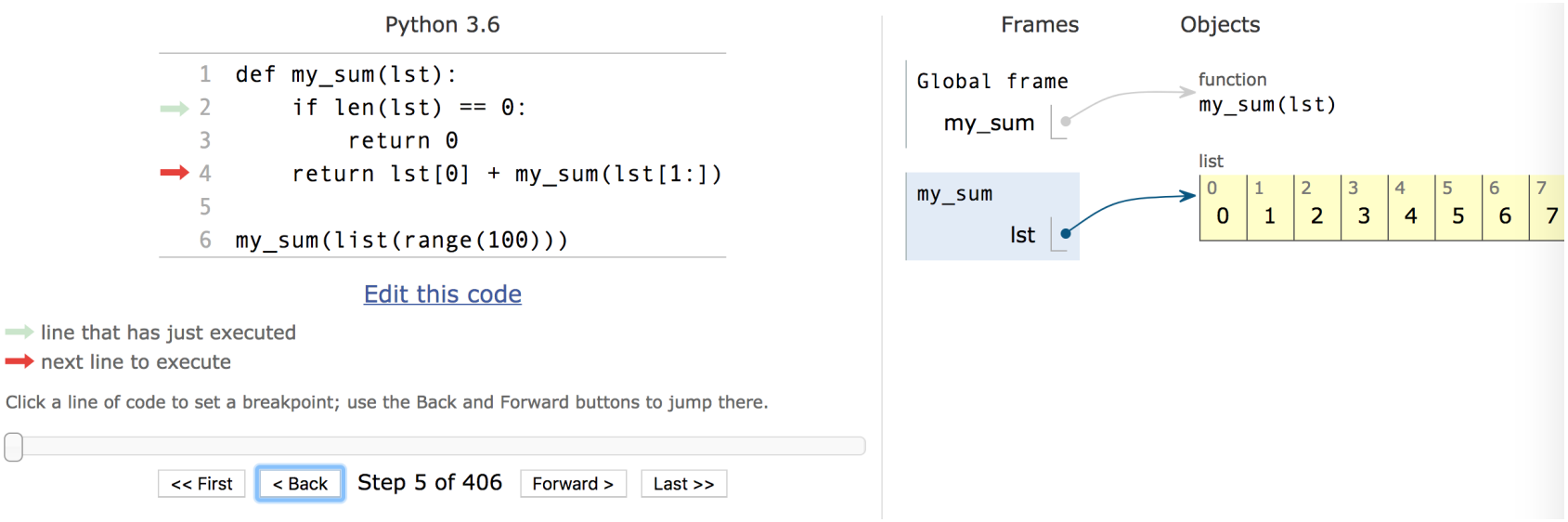

Figure 1: Python call stack: `my_sum` is called from the global frame.

Call Stack - Python Visualizer

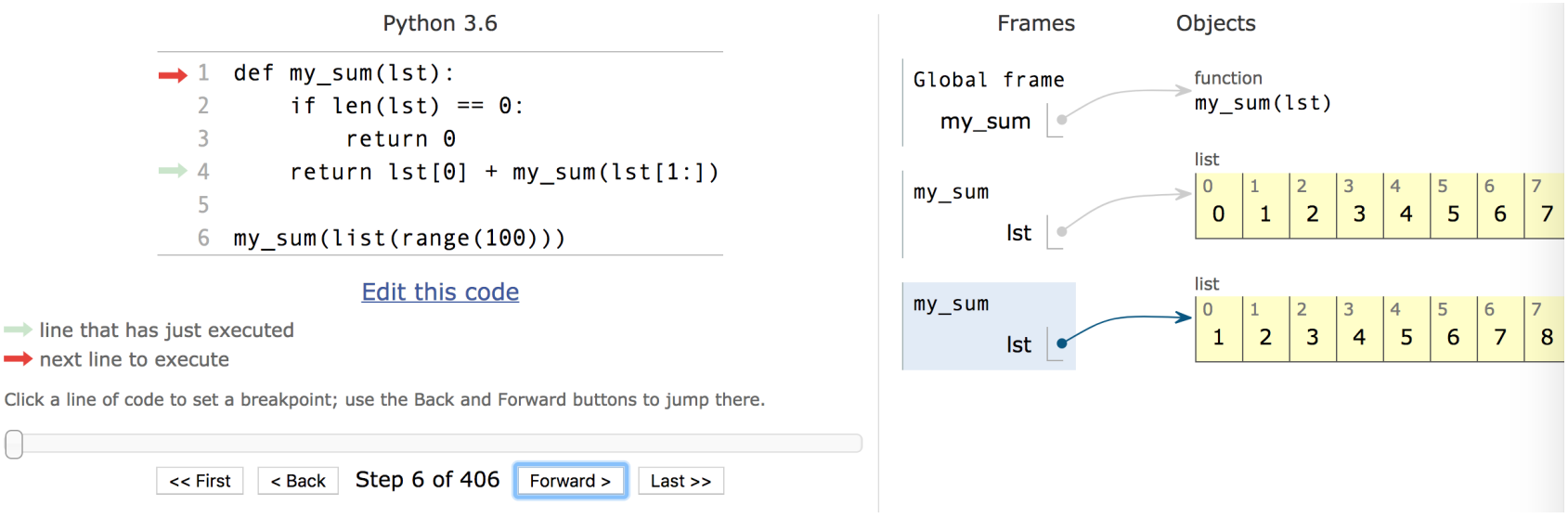

Figure 2: Python call stack: `my_sum` is called from itself.

Call Stack - Python Visualizer

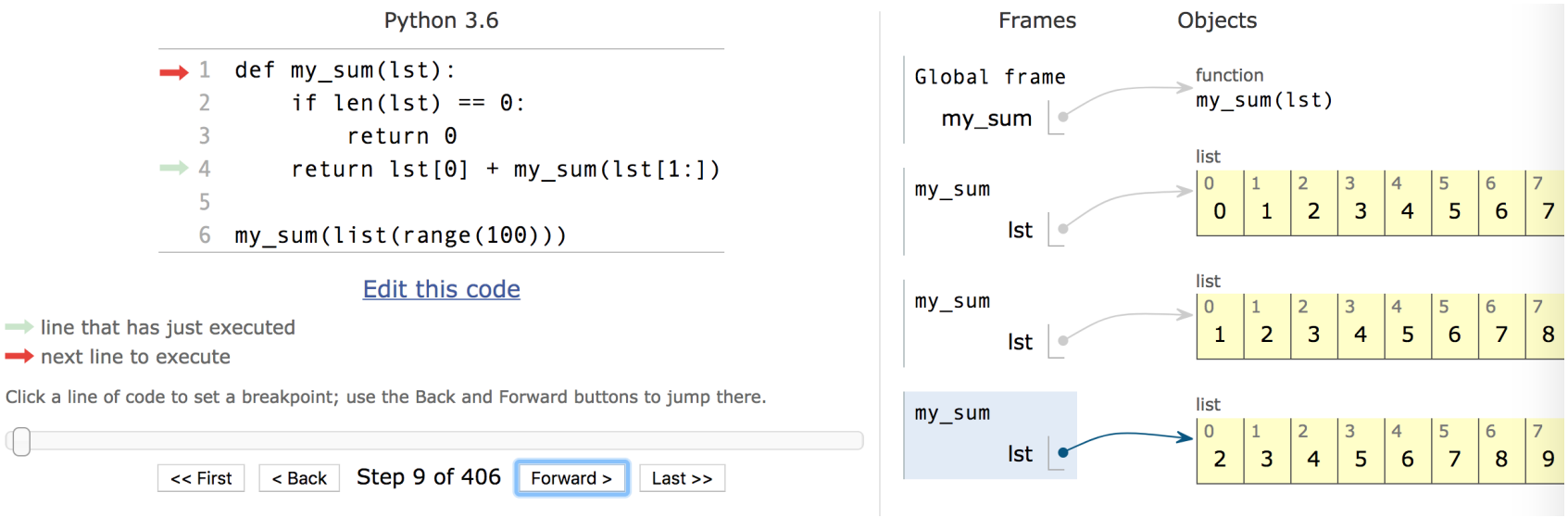

Figure 3: Python call stack: `my_sum` is called from itself, again.

Difficulty with Recursion

In Python:

- Slower than using a loop, because we need to manage the call stack
- Stack overflows

In Racket/Haskell:

- Similar issue, however we can fix both issues at the same time using **tail recursion**

Similar issue in Racket

```
(define (factorial n)
  (if (equal? n 0)
      1
      (* n (factorial (- n 1)))))
```

If we call `(factorial 5)`, then we compute...

- `(* 5 (factorial 4))`
- `(* 5 (* 4 (factorial 3)))`
- ...

Calling (factorial 5)

- (* 5 (factorial 4))
- (* 5 (* 4 (factorial 3)))
- (* 5 (* 4 (* 3 (factorial 2))))
- (* 5 (* 4 (* 3 (* 2 (factorial 1)))))
- (* 5 (* 4 (* 3 (* 2 (* 1 (factorial 0))))))
- (* 5 (* 4 (* 3 (* 2 (* 1 1)))))
- (* 5 (* 4 (* 3 (* 2 1))))
- (* 5 (* 4 (* 3 2)))
- (* 5 (* 4 6))
- (* 5 24)
- 120

A different method: tail recursion

```
(define (factorial-helper n acc)
  (if (equal? n 0)
      acc
      (factorial-helper (- n 1) (* n acc))))
```

A different method: tail recursion

```
(define (factorial-helper n acc)
  (if (equal? n 0)
      acc
      (factorial-helper (- n 1) (* n acc))))
```

If we call `(factorial-helper 5 1)`, then we compute...

- `(factorial-helper 4 5)`
- `(factorial-helper 3 (* 4 5))`
- `(factorial-helper 3 20)`
- `(factorial-helper 2 60)`
- `(factorial-helper 1 120)`
- `(factorial-helper 0 120)`
- `120`

Tail recursion

Racket (and Haskell) has a way to optimize **tail recursion** via **tail-call elimination**

- When the *last thing* evaluated by a function is a recursive call, there is no reason to remember the current call stack
- Frame of the current procedure is *replace* by the frame of the tail call
- No growth in the call stack

Functions that can be optimized this way are called **tail recursive**.

Non-tail recursive function

This function is **not** tail recursive:

```
#lang racket
(define (sum1 lst)
  (if (empty? lst)
      0
      (+ (first lst)
         (sum1 (rest lst)))))
```

- The addition + is the last thing that happens
- Need to keep the stack frame around to store the value of `(first lst)`
- Need to perform the operation + once the recursive call to `sum1` returns

Tail recursive sum

To make `sum` tail recursive, we add a helper function with an **accumulator**:

```
#lang racket
(define (sum2 lst)
  (sum-helper lst 0))

(define (sum-helper lst acc)
  (if (empty? lst)
      acc
      (sum-helper (rest lst) (+ acc (first lst)))))
```

Compare the runtime of `sum2` to `sum1` — `sum1` is much slower, as it wastes time managing the call stack!

Worksheet (Questions 3-4)

Hint:

- You will usually need a helper function with an additional parameter (an accumulator)
- Your accumulator might be a number, or something else

Higher-Order Functions

Functions

A function abstracts computation over possible values of its parameters:

```
(+ 32 (*   1 (/ 9 5)))
(+ 32 (* 100 (/ 9 5)))
(+ 32 (*  -2 (/ 9 5)))

(lambda (x)
  (+ 32 (* x (/ 9 5))))
```

Here, `x` is an parameter that represents a number.

Abstracting over functions

But what if the *operation* (which is a function) is what's different?

```
(+ 32 (* 100 (/ 9 5)))
(- 32 (* 100 (/ 9 5)))
(* 32 (* 100 (/ 9 5)))
```

Higher-Order Functions

In Racket, Haskell, and Python, functions can be arguments too!

```
(+ 32 (* 100 (/ 9 5)))
(- 32 (* 100 (/ 9 5)))
(* 32 (* 100 (/ 9 5)))

(lambda (x)
  (x 32 (* 100 (/ 9 5))))
```

Higher-Order Functions

- A **higher-order** function is a function that takes one or more functions as parameters, or that returns a function
- The defining feature of functional programming is writing functions that take and/or return functions

Here's a Racket example that applies `g` to `x` and then applies `f` to that result:

```
#lang racket

(define (compose f g x)
  (f (g x)))

(compose sqr (lambda (x) (+ x 1)) 3) ; 16
```

Recursive Higher-Order List Functions

- Higher-order list functions abstract the details of how a list is processed
- They help us simplify recursive code

We will discuss:

1. `map`
2. `filter`
3. `foldl`

map

`(map f lst)`: Return a new list by applying function `f` to each element of `lst`

```
> (map (lambda (x) (* x 3)) (list 1 2 3 4))
'(3 6 9 12)
```

An iterative `map` looks like:

```
new_lst = []
for x in lst:
    new_item = f(x)
    new_lst.append(new_item)
```

filter

`(filter pred lst)`: return a new list consisting of the elements of `lst` on which `pred` returns `#t`

```
> (filter (lambda (x) (> x 1)) (list 4 -1 0 15))
'(4 15)
```

An iterative `filter` looks like:

```
new_lst = []
for x in lst:
    if pred(x):
        new_lst.append(x)
```

`foldl`

- `map` and `filter` are **accumulator patterns**: they apply a function to each value of a list and accumulate the results
- `foldl` is a more general accumulation pattern
- In addition to a list, `foldl` takes an initial value, and a function that updates this initial value by using each list element
- `map` and `filter` return a list; `foldl` can return a value of any type!

```
(foldl combine init lst)
```

foldl...

`foldl` works slightly differently in Racket and Haskell!

Racket: `4-(3-(2-(1-0)))`

```
> (foldl - 0 (list 1 2 3 4))
2
```

Haskell: `(((0-1)-2)-3)-4`

```
> foldl (-) 0 [1, 2, 3, 4]
-10
```

`foldl...`

`foldl combine init lst`

An iterative (Racket) `foldl` looks like:

```
acc = init
for x in lst:
    acc = combine(x, acc)
```

Worksheet Q5

An iterative (Racket) `foldl` looks like:

```
acc = init
for x in lst:
    acc = combine(x, acc)
```

Functions Returning Functions

Higher-order functions can also **return** a function!

```
(define (make-adder x)
  (lambda (y) (+ x y)))
(make-adder 10) ; #<procedure>

(define add-10 (make-adder 10))
add-10      ; #<procedure>
(add-10 3) ; 13
```

The Higher-Order Function: `apply`

- Racket's `apply` function takes a function `f` and a list of arguments, and applies each element of the list as a parameter to `f`

```
> (define evens (filter even?
    (list 1 2 3 4 5 6 7 8 9 10)))
> evens
'(2 4 6 8 10)
> (apply + evens)
30
```

- Haskell has the `$` operator for function application, but it does not "unpack" a list as does `apply`!

Currying

Currying

```
big lst = filter (\ x -> 5 < x) lst
```

- Notice that the anonymous function is just the < function with its first parameter "filled in"
- Currying allows us to produce new functions by supplying only *some* of a function's parameters

```
big lst = filter ((<) 5) lst
```

Python `functools.partial`

Currying in Haskell

- Haskell **automatically** curries functions!

```
Prelude> add x y = x + y
Prelude> add2 = add 2
Prelude> add2 3
5
```

Abstract Syntax Trees

Need to talk about AST for the exercise.