CSC324 Principles of Programming Languages

Lecture 2

September 21/22, 2020

Announcement

- Ex1 unit tests are released on Markus
  - Don't remove required code; make sure your code runs
  - Re-run Sep 23, 10pm
- Ex2 and Ex3 are released; we'll discuss both during this week's labs
  - Minor fix to ex2: `apply-fns` does not have to be tail-recursive
  - List functions like `append`, `reverse` are ok to use
  - No HO list functions like `map`, `filter`, `foldl`
- Test 1 is next week during the labs (more on this. . .)

Test 1 logistics

Test 1 will be a Quercus Quiz that will be available during the lab next week.

- **You must write the test with your lab section!** If you need to write at a different time, or if you require accessibility accommodation, email Lisa by Sep 25.
- 30 minutes in length
- You can begin writing between 5-15 min past the hour (e.g. 11:05-11:15) and must finish writing by 45 min past the hour (e.g. 11:45)
- **You may not communicate with any other person while writing this test**
  - Being on email, Piazza, BbCollaborate, etc is an academic offense
- We will **not** be answering questions!
- Don't talk about the test until the next day

Test 1 topics

The test is open book, and will be graded out of 15. It will have 10 multiple choice questions, 1 coding questions, and 1 short-answer question.

- Covers the materials from weeks 1-2
- Open book, but you may not communicate with another person
  - Academic integrity is a matter of pride and *integrity*
  - Test questions may reference materials from the notes, quizzes, and *your* prior course work to verify your identity
- The learning objectives covered are listed here:
  - https://q.utoronto.ca/courses/176050/pages/test-1-next-week

What happens if...

- I have a question about one of the test questions?
  - Complete the question to the best of your abilities
  - We won't answer any questions during the test because you can't communicate.
- I miss the test due to an emergency?
  - Weight of first missed test shifted to the exam
  - Subsequent missed tests will be replaced by an oral makeup test
- I am disconnected from the internet?
  - Let Lisa know **as soon as you can** by email
  - Document the issue: the time of outage, time that you can reconnect
  - Take a picture of the Quercus quiz not loading, along with the current time
- There are other anomalies?
  - Complete the test to the best of your ability, and let Lisa know by email after the test

Overview

- **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
- **Higher-order list functions**: `map`, `filter`, `foldl`

We will assume that you have read week 2 notes

Pattern Matching

Value Pattern-Matching

Pattern-matching: a way to specify different function bodies for different inputs.

Value pattern-matching: specify different function bodies for different input *values*.

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

Value Pattern-Matching

Pattern-matching: a way to specify different function bodies for different inputs.

Value pattern-matching: specify different function bodies for different input *values*.

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

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

Structural Pattern-Matching

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

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

Structural Pattern-Matching

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

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

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

Racket Pattern Matching

```
; Value pattern matching
(define/match (add1_lst lst)
  [('()) '()]
  [(lst) (cons (+ 1 (first lst)) (add1_lst (rest lst)))])

; Structural pattern matching
(define/match (add1_lst lst)
  [('()) '()]
  [((cons x xs)) (cons (+ 1 x) (add1_lst xs))])
```

Breakout Group [10 minutes]

Complete Exercise 1, 2, 3 in the worksheet pasted in the chat.

Exercise 1

Rewrite this function use value pattern matching in Haskell.

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

Exercise 1

Rewrite this function use value pattern matching in Haskell.

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

--solution
f 1 = "one"
f 2 = "two"
f 3 = "three"
f _ = "error"
```

Exercise 2

Exercise: Rewrite this function use pattern matching

```
(define (contains elem lst)
  (if (empty? lst)
      #f
      (or (equal? elem (first lst))
          (contains elem (rest list)))))
```

Exercise 2

Exercise: Rewrite this function use pattern matching

```
(define (contains elem lst)
  (if (empty? lst)
      #f
      (or (equal? elem (first lst))
          (contains elem (rest list)))))
```

```
; solution
(define/match (contains elem lst)
   [(elem '()) #f]
   [(elem (cons x xs)) (or (equal? elem x)
                           (contains elem xs))])
```

Exercise 3

This Haskell function list_sum adds up the elements of a list.
Rewrite the function list-sum in Racket.

```
list_sum [] = 0
list_sum (x:xs) = x + list_sum xs
```

Exercise 3

This Haskell function list_sum adds up the elements of a list. Rewrite the function list-sum in Racket.

```
list_sum [] = 0
list_sum (x:xs) = x + list_sum xs
```

```
; solution
(define/match (list-sum lst)
   [('()) 0]
   [((cons x xs)) (+ x (list-sum xs))])
```

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: `my_sum([1, 2, 3, 4])`

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

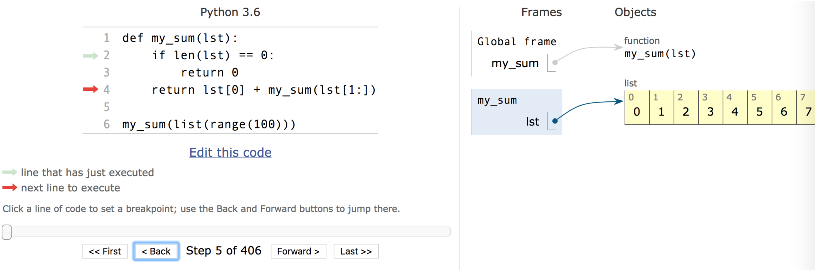

Figure 1: Python call stack: `my_sum` is called from the global frame.

Call Stack - Python Visualizer

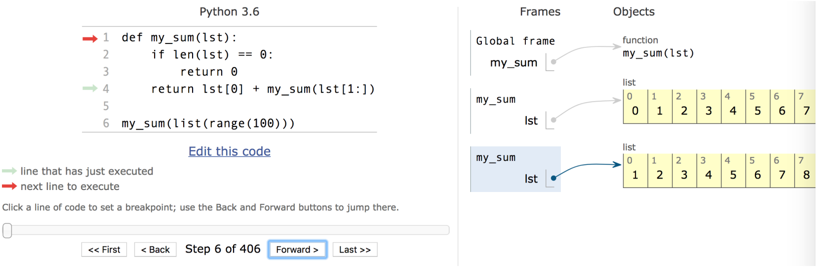

Figure 2: Python call stack: `my_sum` is called from itself.

Call Stack - Python Visualizer

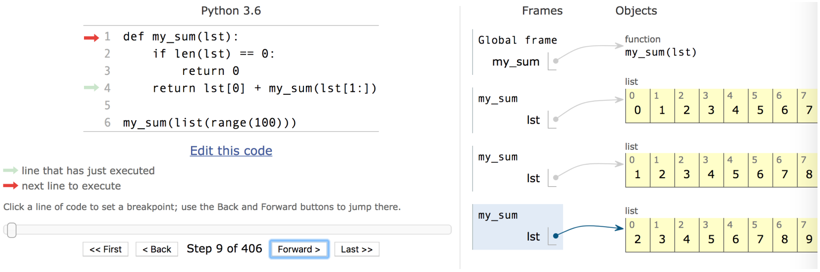

Figure 3: Python call stack: `my_sum` is called from itself, again.

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, but functional languages like Racket/Haskell has an optimization for **tail recursion** that fixes both issues

Similar issue in Racket

```
; from earlier
(define/match (list-sum lst)
  [('()) 0]
  [((cons x xs)) (+ x (list-sum xs))])
```

If we call `(list-sum '(1 2 3 4 5))`, then we compute. . .

- `(+ 1 (list-sum '(2 3 4 5)))`
- `(+ 1 (+ 2 (list-sum '(3 4 5))))`
- . . .

Calling (list-sum '(1 2 3 4 5))

- (+ 1 (list-sum '(2 3 4 5)))
- (+ 1 (+ 2 (list-sum '(3 4 5))))
- (+ 1 (+ 2 (+ 3 (list-sum '(4 5)))))
- (+ 1 (+ 2 (+ 3 (+ 4 (list-sum '(5))))))
- (+ 1 (+ 2 (+ 3 (+ 4 (+ 5 (list-sum '()))))))
- (+ 1 (+ 2 (+ 3 (+ 4 (+ 5 0)))))
- (+ 1 (+ 2 (+ 3 (+ 4 5))))
- (+ 1 (+ 2 (+ 3 9)))
- (+ 1 (+ 2 12))
- (+ 1 14)
- 15

A different method: tail recursion

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

A different method: tail recursion

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

If we call `(list-sum-helper '(1 2 3 4 5) 0)`, then we compute...

- `(list-sum-helper '(2 3 4 5) 1)`
- `(list-sum-helper '(3 4 5) 3)`
- `(list-sum-helper '(4 5) 6)`
- `(list-sum-helper '(5) 10)`
- `(list-sum-helper '() 15)`
- `15`

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 eecursive 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:

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

- The addition `*` is the last thing that happens
- Need to keep the stack frame around to store the value of `n`
- Need to perform the operation `*` once the recursive call to `factorial` returns

To make `factorial` tail recursive, we add a helper function with an **accumulator**.

Breakout Group [10 minutes]

Complete Exercise 4, 5, 6 in the same document.

Exercise 4

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

Exercise 4

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

; solution
(define (factorial n) (fac-helper n 1))
(define/match (fac-helper n acc)
  [(1 acc) acc]
  [(n acc) (fac-helper (- n 1) (* n acc))])
```

Exercise 5

```
list_double [] = []
list_double (x:xs) = (x*2):(list_double xs)
```

Exercise 5

```
list_double [] = []
list_double (x:xs) = (x*2):(list_double xs)

-- solution
list_double lst = h lst []
h []     acc = reverse acc
h (x:xs) acc = h xs ((2*x):acc)
```

Example 6

Are these functions tail recursive?

```
-- Haskell
collatz 1 = 1
collatz n =
    if (mod n 2) == 0
        then collatz (quot n 2)     -- integer division
        else collatz ((n * 3) + 1)
```

```
; Racket
(define (collatz n)
  (cond
    [(equal? n 1)             1]
    [(equal? (modulo n 2) 0) (collatz (/ n 2))]
    [else                    (collatz (+ (* n 3) 1))]))
```

Example 6

Are these functions tail recursive?

```
-- Haskell
collatz 1 = 1
collatz n =
    if (mod n 2) == 0
        then collatz (quot n 2)     -- integer division
        else collatz ((n * 3) + 1)
```

```
; Racket
(define (collatz n)
  (cond
     [(equal? n 1)               1]
     [(equal? (modulo n 2) 0) (collatz (/ n 2))]
     [else                    (collatz (+ (* n 3) 1))]))
```

Yes!

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:

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

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

Example 1. Type in the chat...

What does this function return?

```
mystery f g = \x -> f (g x)
```

Example 1. Type in the chat...

What does this function return?

```
mystery f g = \x -> f (g x)
```

Side Note: Alternative Haskell syntax to avoid brackets

```
mystery f g = \x -> f $ g x
```

Example 2. Type in the chat...

What does `(f 3)` return?

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

(define (g h) (h 1))
```

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

Apply a function to every element of a list

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

```
Prelude> map (\x -> x * 3) [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)
```

Type in the chat. . .

What is the value of this Racket expression:

```
(map (lambda (x) (equal? x 5)) '(3 4 5 6))
```

Type in the chat...

What is the value of this Racket expression:

```
(map (lambda (x) (equal? x 5)) '(3 4 5 6))
```

What is the value of `z` in this Haskell expression:

```
z = map (\x -> (\y -> x)) [3, 4, 5]
```

filter

Return a new list consisting of the elements on which the predicate returns true

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

```
Prelude> filter (\x -> x > 1) [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!

```
# Racket:
(foldl combine init lst)

; Haskell:
foldl combine init lst
```

`foldl...`

`foldl combine init lst`

`foldl` works slightly differently in Racket and Haskell!

An iterative (Racket) `foldl` looks like:

```
acc = init
for x in lst:
    acc = combine(x, acc)    # racket version
    # acc = combine(acc, x)  # haskell version
```

foldl...

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

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

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

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

Exercise

Rewrite this function using `map`, `filter`, or `foldl`.

```
(define/match (double-odd lst)
  [('()) '()]
  [((cons x xs)) (if (odd? x)
                     (cons (* 2 x) (double-odd xs))
                     (double-odd xs))])
```

Exercise

Rewrite this function using `map`, `filter`, or `foldl`.

```
(define/match (double-odd lst)
  [('()) '()]
  [((cons x xs)) (if (odd? x)
                     (cons (* 2 x) (double-odd xs))
                     (double-odd xs))])
```

```
(define (double-odd lst)
  (map (lambda (x) (* x 2)) (filter odd? lst)))
```

Currying

Currying

```haskell
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

```haskell
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
???
```

Sectioning

Sectioning lets us fix the left *or* right parameter of a binary operator

```
Prelude> add2 = (2 +) -- fix left arg
Prelude> add2 10
???
Prelude> sub2 = (- 2) -- fix right arg
Prelude> sub2 10
???
```

Currying in Haskell

Recall this function in Haskell. How can we use currying to rewrite this function?

```
compose f g = \x -> f (g x)
```

Currying in Haskell

Recall this function in Haskell. How can we use currying to rewrite this function?

```
compose f g = \x -> f (g x)
```

```
Prelude> compose f g x = f (g x)
Prelude> h = compose (1 +) (2 *)
Prelude> h 5
???
```

What to do next

1. Complete week 2 quiz; ask questions on Piazza
2. Attend the labs this week to learn about ex2/ex3
3. Exercise 2 due Saturday 10pm
4. Read week 3 notes and attempt week 3 quiz next week
5. Prepare for test 1 next Tuesday