

CSC324: Principles of Programming Languages

Lecture 3

Friday, 15 May, 2020

Announcements

- Exercise 1 marked soon!
- Exercise 2 posted soon!
- No lab session on Monday (happy Victoria Day!)
 - Please still work through Lab 2 next week, when it is posted. If you have questions that you would have asked during the lab session, come to my office hours.

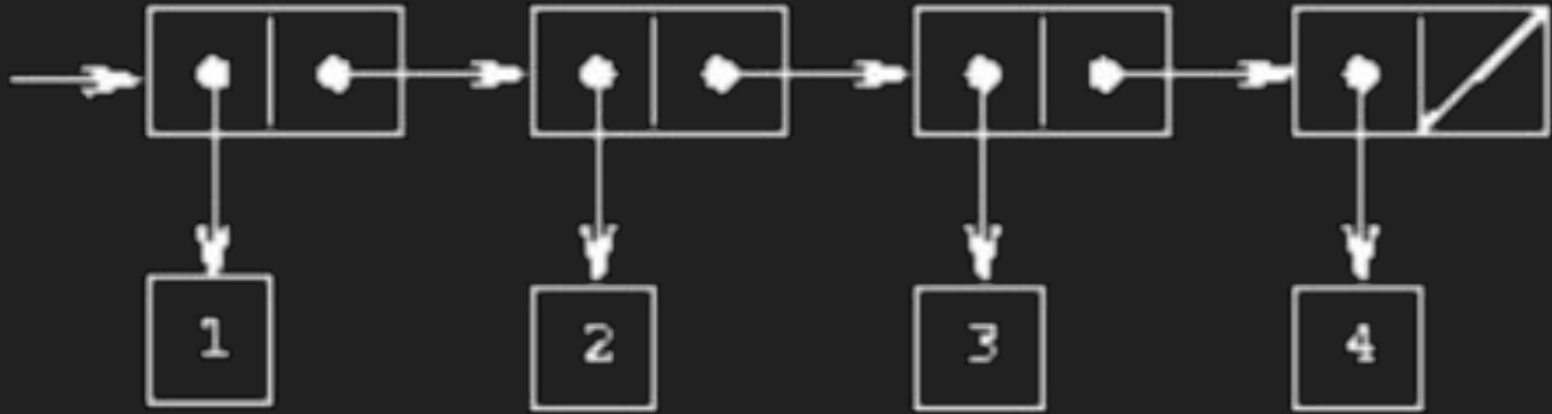
Lists and Structural Recursion

Last time, we saw a correspondence between a recursive datatype (the AST) and a grammar that the AST satisfies

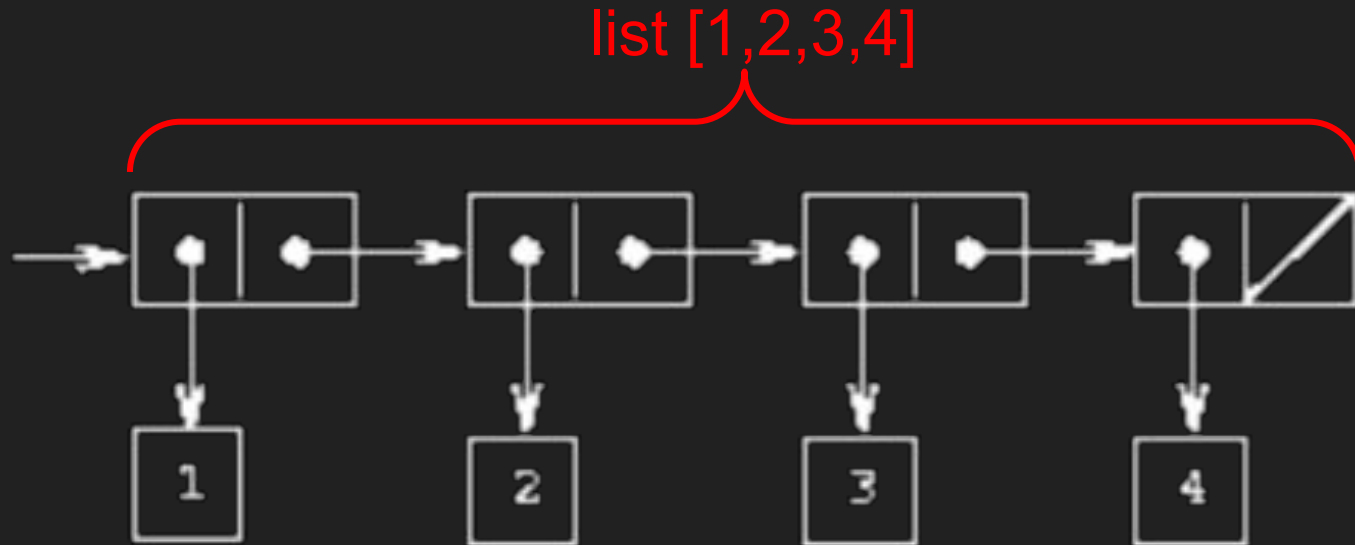
Let's consider another relationship that a recursive datatype has: with a function that performs some operation over that datatype

We'll do this with the most fundamental data structure: a singly-linked list.

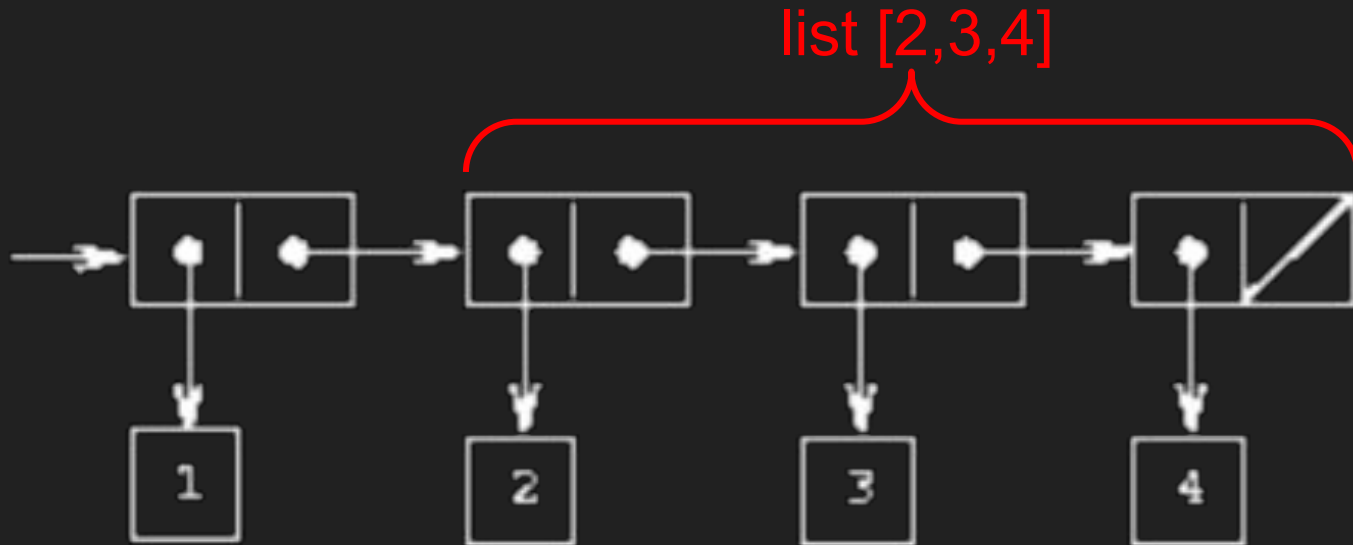
The List datatype



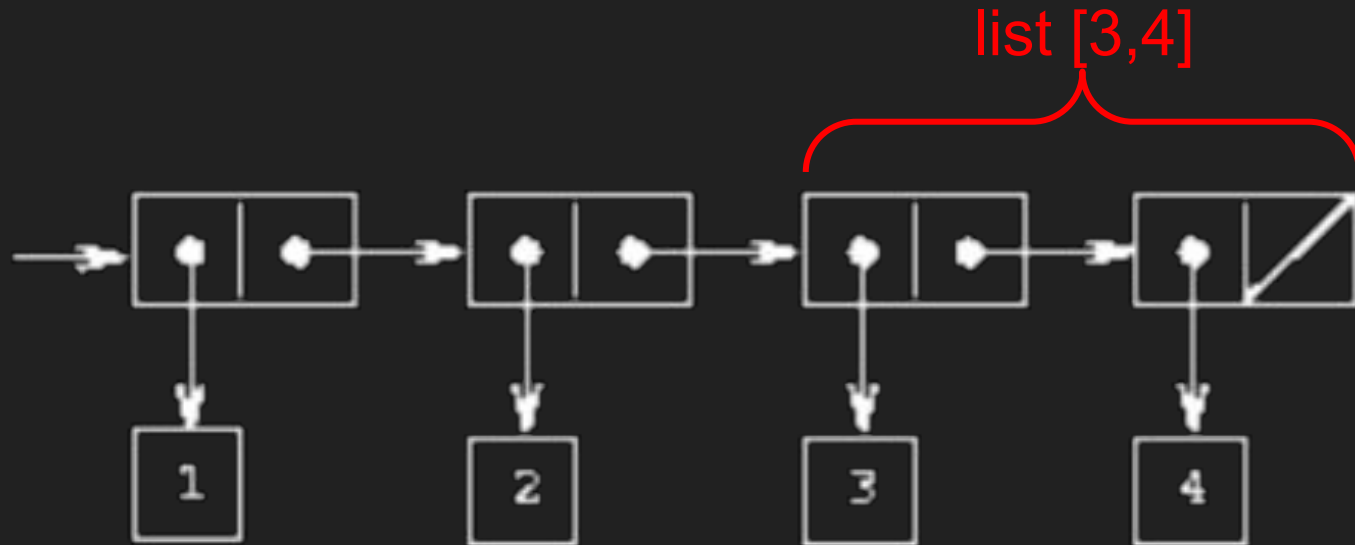
Life lesson: look for recursive structure!



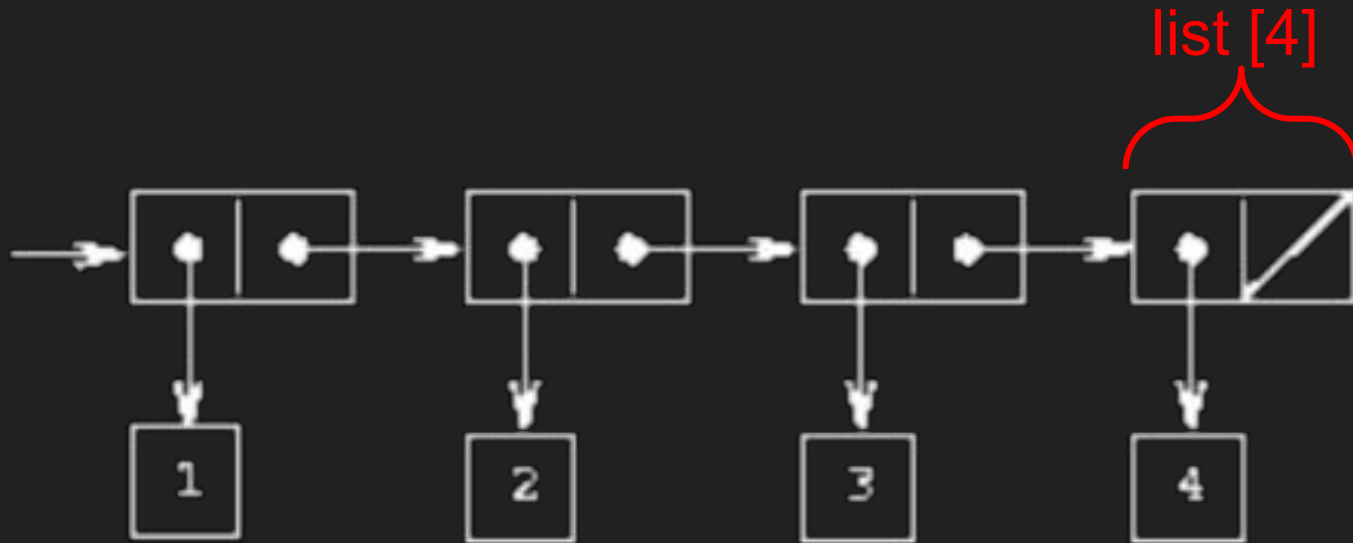
Life lesson: look for recursive structure!



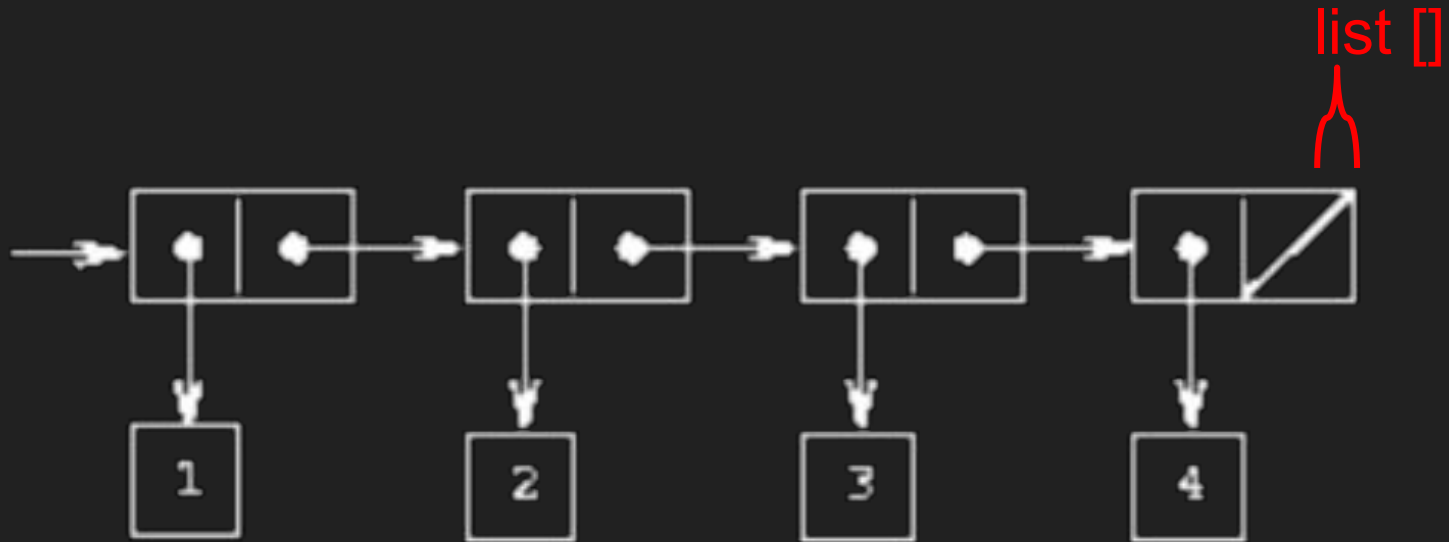
Life lesson: look for recursive structure!



Life lesson: look for recursive structure!



Life lesson: look for recursive structure!



Life lesson: look for recursive structure!

A **list** is:

- The empty list;
- A value prepended onto a **list**

The empty list `[]` is a list

4 prepended onto `[]` is a list

3 prepended onto (4 prepended onto `[]`) is a list

...

List construction in Racket

- Empty list: '() (note: the single quote before () is significant!)
- An element x prepended onto a list xs: **(cons x xs)**
- The list containing 1,2,3: (list 1 2 3)

List construction in Haskell

- Empty list: []
- An element x prepended onto a list xs : $x : xs$
- The list containing 1,2,3: [1,2,3]

An iterative & mutable approach to operating on lists

```
1 def find_in_list(l, val):  
2     while l is not None:  
3         if l.value == val:  
4             return True  
5         l = l.next  
6     return False  
7
```

An iterative & mutable approach to operating on lists

```
1 def list_length(l, val):  
2     size = 0  
3     while l is not None:  
4         size += 1  
5         l = l.next  
6     return size  
7  
8
```

An iterative & mutable approach to operating on lists

- Because we can't rebind the "current node" variable in a loop, we need an alternate way of walking the list
- Back to the drawing board: what strategy can we take? Remember our life lesson...

```
1 def find_in_list(l, val):  
2     while l is not None:  
3         if l.value == val:  
4             return True  
5         l = l.next  
6     return False  
7
```

```
1 def list_length(l, val):  
2     size = 0  
3     while l is not None:  
4         size += 1  
5         l = l.next  
6     return size  
7
```

find_in_list, defined by the recursive datatype

A list is:

- The empty list '()
- ...or... (cons x xs):
 - an element x
 - prepended on list xs

find_in_list:

- The empty list will never hold `val`
- ...or...
 - val is equal to x
 - val is found in xs

```
1 def find_in_list(l, val):  
2     while l is not None:  
3         if l.value == val:  
4             return True  
5         l = l.next  
6     return False  
7
```


list_length, defined by the recursive datatype

A list is:

- The empty list '()
- ...or... (cons x xs):
 - an element x
 - prepended on list xs

find_in_list:

- The empty list has a length of zero
- ...or... the list's length is one more than
 - the list length of xs

```
1 def list_length(l, val):
2     size = 0
3     while l is not None:
4         size += 1
5         l = l.next
6     return size
7
8
```

Structural recursion on lists

A list is therefore:

- The empty list
- ...or...
 - an element x
 - prepended on list xs

A function that operations on a list is therefore:

- A base case on the empty list
- ..or...
 - Some operation on x
 - combined with a recursive call on xs

```
(define (list-op-template l)
  (if (empty? l) ...
      (... (car l)
           (list-op-template (cdr l))))))
```

Structural recursion on (the natural) numbers?

Remember this function from last time?

```
#lang racket
(require rackunit)

(define (slow-add m n)
  (if (= n 0)
      m
      (slow-add (+ m 1) (- n 1))))

(check-eq? (slow-add 5 5) 10)
(check-eq? (slow-add 0 6) 6)
(check-eq? (slow-add 6 0) 6)
```

Structural recursion on (the natural) numbers?

If you looked at the optional reading on datatypes in the lambda calculus, this might give you some intuition for why they defined numbers as they did!

```
(define (nat-num-op-template n)
  (if (= n 0) ...
      (... (nat-num-op-template (- n 1)))))
```

So, we saw:

- We saw how a **recursively-defined** list datatype has a **direct correspondence to writing a structurally-recursive function that operates on it**
- How to put lists together in Racket using `(cons x xs)` and in Haskell using `x : xs`
- How to take lists apart using `car` and `cdr` in Racket, and `head` and `tail` in Haskell

Counting like a troll

*"Everyone knows trolls can't even count up to four!"**

**In fact, trolls traditionally count like this: one, two, three, many, and people assume this means they can have no grasp of higher numbers. They don't realise that many can BE a number."*

Terry Pratchett, Men at Arms



Counting like a troll

```
1 def num_to_troll(n):  
2     if n == 1:  
3         return "one"  
4     elif n == 2:  
5         return "two"  
6     elif n == 3:  
7         return "three"  
8     elif n == 4:  
9         return "many"  
10    else:  
11        return "lots"  
12  
13
```



Counting like a troll

```
(define (num-to-troll n)
  (if (= n 1)
      "one"
      (if (= n 2)
          "two"
          (if (= n 3)
              "three"
              (if (= n 4)
                  "many"
                  "lots"))))))
```



Conding like a troll

```
(define (num-to-troll n)
  (cond [(= n 1) "one"]
        [(= n 2) "two"]
        [(= n 3) "three"]
        [(= n 4) "many"]
        [else "lots"])))
```

This works, but it's slightly annoying and redundant that every left-hand side of every cond clause is comparing n.



Matching like a troll

```
(define (num-to-troll n)
  (match* (n)
    [(1) "one"]
    [(2) "two"]
    [(3) "three"]
    [(4) "many"]
    [(_) "lots"])))
```



Structural recursion with cond

Can we write our list template using cond rather than if?

```
(define (list-op-template l)
  (cond ((empty? l) ...)
        ((... (car l)
               (list-op-template (cdr l))))))
```

Structural recursion with cond

This works, but... we have a cond with two "cases", one for the empty list and one that's an element prepended onto a list. Can we make the code below look more like the data definition on the right?

A list is:

- *The empty list '()*
- *...or... (cons x xs):*
 - *an element x*
 - *prepending on list xs*

```
(define (list-op-template l)
  (cond [(empty? l) ...]
        [else (... (car l)
                     (list-op-template (cdr l)))]))
```

Structural recursion with match

Structural recursion

AND

structural decomposition!

A list is:

- *The empty list '()*
- *...or... (cons x xs):*
 - *an element x*
 - *prepended on list xs*

```
(define (list-op-template l)
  (match* (l)
    [('()) ...]
    [((cons x xs)) (... x
                        (list-op-template xs))]))
```

Structural decomposition with match

car and cdr (and head and tail) are all functions that decompose lists.

```
> (car '(1 2 3))  
1  
> (cdr '(1 2 3))  
'(2 3)  
>
```


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

Structural decomposition

We used them in our structural recursion to extract the first and rest of the list on the right-hand side of a cond clause.

Pattern matching makes this cleaner by doing so on the left-hand side.

```
(define (list-op-template l)
  (cond [(empty? l) ...]
        [else (... (car l)
                    (list-op-template (cdr l)))]))
```

Two red arrows are present. The first arrow points from the right towards the expression `(car l)` in the `[else (... (car l) ...)]` clause. The second arrow points from the right towards the expression `(list-op-template (cdr l))` in the same clause.

Structural decomposition

```
(define (list-op-template l)
  (cond [(empty? l) ...]
        [else (... (car l)
                     (list-op-template (cdr l)))]))
```

```
(define (list-op-template l)
  (match* (l)
    [('()) ...]
    [((cons x xs)) (... x
                        (list-op-template xs))]))
```


Structural decomposition

```
(define (list-op-template l)
  (cond [(empty? l) ...]
        [else (... (car l)
                     (list-op-template (cdr l)))]))
```

```
(define (list-op-template l)
  (match* (l)
    [('()) ...]
    [(cons x xs) (... x
                       (list-op-template xs))]))
```

The algebraic laws for lists

For all x and y ,

- $(\text{car } (\text{cons } x \text{ } xs)) \equiv x$ (*the law of car*)
- $(\text{cdr } (\text{cons } x \text{ } xs)) \equiv xs$ (*the law of cdr*)

Transforming the cond-based template to the match-based template is completely intuitive, if you think of it as performing **algebraic manipulations**

Tail calls

Consider again our implementation of list-length, in Haskell for a change.

A fatal flaw lurks within (assuming our current evaluation-by-substitution model)...

```
foo.hs
1 list_length :: [a] -> Int
2 list_length [] = 0
3 list_length (x:xs) = 1 + list_length xs
4
```

Tail calls

foo.hs

```
1 list_length :: [a] -> Int
2 list_length [] = 0
3 list_length (x:xs) = 1 + list_length xs
4
```

```
list_length [1,2,3,4,5]
```

Tail calls

foo.hs

```
1 list_length :: [a] -> Int
2 list_length [] = 0
3 list_length (x:xs) = 1 + list_length xs
4
```

```
list_length [1,2,3,4,5]
```

```
1 + list_length [2,3,4,5]
```

Tail calls

foo.hs

```
1 list_length :: [a] -> Int
2 list_length [] = 0
3 list_length (x:xs) = 1 + list_length xs
4
```

```
list_length [1,2,3,4,5]
1 + list_length [2,3,4,5]
1 + 1 + list_length [3,4,5]
```

Tail calls

foo.hs

```
1 list_length :: [a] -> Int
2 list_length [] = 0
3 list_length (x:xs) = 1 + list_length xs
4
```

```
list_length [1,2,3,4,5]
1 + list_length [2,3,4,5]
1 + 1 + list_length [3,4,5]
1 + 1 + 1 + list_length [4,5]
1 + 1 + 1 + 1 + list_length [5]
1 + 1 + 1 + 1 + 1
```

Tail calls

foo.hs

```
1 list_length :: [a] -> Int
2 list_length [] = 0
3 list_length (x:xs) = 1 + list_length xs
4
```

```
list_length [1,2,3,4,5]
1 + list_length [2,3,4,5]
1 + 1 + list_length [3,4,5]
1 + 1 + 1 + list_length [4,5]
1 + 1 + 1 + 1 + list_length [5]
1 + 1 + 1 + 1 + 1
1 + 1 + 1 + 2
1 + 1 + 3
1 + 4
5
```


$O(n)$ space needed to evaluate `list_length`!

- The size of the expression to be evaluated is proportional to the size of the input list!
- Equivalent to the number of stack frames being proportional to the size of the input list in an imperative language

```
>>> def list_length(l):
...     if l == []:
...         return 0
...     return 1 + list_length(l[1:])
...
>>> list_length([1,2,3])
3
>>> list_length(list(range(10000)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in list_length
  File "<stdin>", line 4, in list_length
  File "<stdin>", line 4, in list_length
  [Previous line repeated 995 more times]
  File "<stdin>", line 2, in list_length
RecursionError: maximum recursion depth exceeded in comparison
>>> █
```

slow-add, revisited

Notice with slow-add that the result of the function call, after 1 fn application,

`(slow-add 2 3)`

is exactly the same as the result of the function call in the recursive case,

`(slow-add 3 2)`

There is no intermediary expression to evaluate that depends on the return value of the recursive call.

```
#lang racket
(require rackunit)

(define (slow-add m n)
  (if (= n 0)
      m
      (slow-add (+ m 1) (- n 1))))

(check-eq? (slow-add 2 3) 5)
(check-eq? (slow-add 3 2) 5)
(check-eq? (slow-add 4 1) 5)
(check-eq? (slow-add 5 0) 5)
```

Tail call

Definition: a **tail call** is a function call that appears as the final expression in a recursive definition. Such a function is said to be in **tail form**.

If a function is in tail form, no larger expression needs to be evaluated after the recursive call completes, so the space requirements of evaluating the function is $O(1)$

```
(slow-add 2 3)
(slow-add 3 2)
(slow-add 4 1)
(slow-add 5 0)
```

Converting a function into tail-form


Note that `slow-add` benefitted from the `m` argument serving as an **accumulator**, where, rather than building up a huge final expression to be evaluated, the final result is computed one step at a time in constant space.

How can we take advantage of this for a function that is not in tail-form?

```
#lang racket
(require rackunit)

(define (slow-add m n)
  (if (= n 0)
      m
      (slow-add (+ m 1) (- n 1))))

(check-eq? (slow-add 2 3) 5)
(check-eq? (slow-add 3 2) 5)
(check-eq? (slow-add 4 1) 5)
(check-eq? (slow-add 5 0) 5)
```



Converting a function into tail-form

As in the previous lecture, we:

- add an additional accumulating parameter to the function;
- change the base case to return the accumulator;
- move the recursive call into tail position, updating the accumulator
- Wrap the function with the accumulator in an outer one, that calls it with the starting acc value

From iteration to recursion, round two

We need to change fact to consume an extra **accumulating** parameter for `n_fact`.

It's common for the base case of functions that use accumulators to just return the accumulator itself.

```
1 def fact(n, n_fact):
2     if n == 1:
3         return n_fact
4         return fact(n-1, n * n_fact)
5
6 assert(fact(1) == 1)
7 assert(fact(2) == 2)
8 assert(fact(3) == 6)
9 assert(fact(20) == 2432902008176640000)
10 print("all tests passed!")
11
12
```

Are we done?



Lieutenant Colonel Hot Take

@cemerick



decent lisp compilers are always so slow, that's one of their cons

[Tweet übersetzen](#)

12:18 nachm. · 27. Apr. 2020 · [TweetDeck](#)

2 Retweets **30** „Gefällt mir“-Angaben
