

CSC324: Principles of Programming Languages

Lecture 4

Wednesday, 20 May 2020

A few words on Exercise 1

Generally speaking, very well done!

If you lost points on celsius-to-fahrenheit *and* you multiplying by 1.8 rather than ($\div 5$), email me. (The test suite failed to take integer vs floats into account, and I should have caught and fixed all cases, but let me know if I missed you.)

if copies: four copies	pass	1	1
(celsius-to-fahrenheit 38)	fail	0	1
(celsius-to-fahrenheit 37)	fail	0	1
(celsius-to-fahrenheit -100)	pass	1	1
(celsius-to-fahrenheit 14)	fail	0	1
if not all functions defined	fail	0	0

A few words on Exercise 1

The most commonly-failed test:

```
(num-many-evens (list (list)))
```

"A list of one list, which is empty" - a valid list of (list of integers)!

Wrapping up tail calls

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.

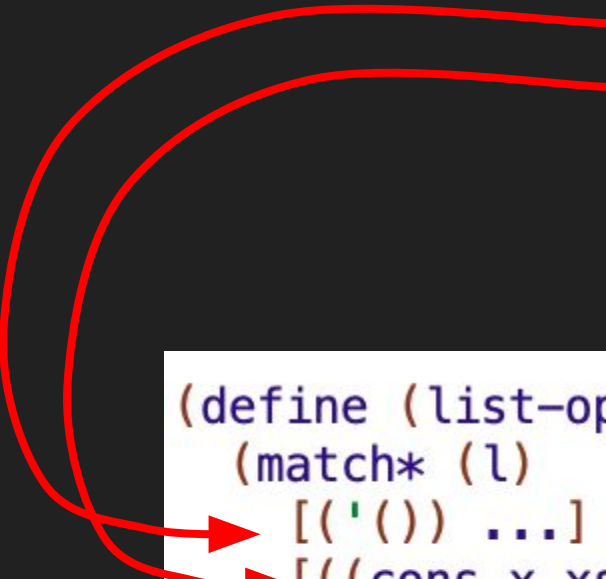
```
1 def fact(n, n_fact):
2     if n == 1:
3         return n_fact
4         return fact(n-1, n * n_fact)
5
6 assert(fact(1) == 1)
7 assert(fact(2) == 2)
8 assert(fact(3) == 6)
9 assert(fact(20) == 2432902008176640000)
10 print("all tests passed!")
11
12
```

Are we done?

Revisiting our List data definition

A list is:

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



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


A binary tree is:

- *The empty tree 'nil*
- *...or... (tree left val right)*
 - *left and right are trees*
 - *val is a value*

```
; a binary tree holding values of type T is:  
; the empty list 'nil  
; a (Node left val right), where:  
;   - left and right are binary trees  
;   - val is a value of type T  
(struct Node (left val right) #:transparent)
```

```
; a binary tree holding values of type T is:  
; the empty list 'nil  
; a (Node left val right), where:  
;   - left and right are binary trees  
;   - val is a value of type T  
(struct Node (left val right) #:transparent)
```

```
(define/match (tree-template t)  
  [( 'nil) ...]  
  [((Node l v r)) (... (tree-template l)  
                        v  
                        (tree-template r))])
```

tree depth

The depth of a binary tree is defined as:

- If the tree is empty: ???
- If the tree is nonempty: ???

tree depth

The depth of a binary tree is defined as:

- If the tree is empty: 0
- If the tree is nonempty:
 - the deeper of the two subtrees 'left' and 'right'
 - plus one for the current tree node

Algebraic Data Types

A list is:

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

A binary tree is:

- *The empty tree 'nil*
- *...or... (tree left val right)*
 - *left and right are trees*
 - *val is a value*

Both our list and tree data definitions are **algebraic data types**

```
Prelude> data Tree a = Nil | Tree (List a) a (List a)
```

Read: "We define a datatype Tree of some type A to be ..." (Java: *Tree<A>*)

Algebraic Data Types

A list is:

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

A binary tree is:

- *The empty tree 'nil*
- *...or... (tree left val right)*
 - *left and right are trees*
 - *val is a value*

Both our list and tree data definitions are **algebraic data types**

```
Prelude> data Tree a = Nil | Tree (List a) a (List a)
```

- a **sum type**: a Tree is Nil or a Tree

Algebraic Data Types

A list is:

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

A binary tree is:

- *The empty tree 'nil*
- *...or... (tree left val right)*
 - *left and right are trees*
 - *val is a value*

Both our list and tree data definitions are **algebraic data types**

```
Prelude> data Tree a = Nil | Tree (List a) a (List a)
```

- a **sum type**: a Tree is Nil or a Tree
- a **product type**: A Tree is a List of a and an a and another List of a.

template functions for algebraic data types

For a **sum type**, we have one arm of a match expression for each possible definition in the sum type

Intuition: for some sum type $x \mid y \mid z$, we have to distinguish between x,y,z

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

template functions for algebraic data types

For a **product type**, we destructure all the fields out of the product type.

If a field is the type we're writing the template for (the tail of a list, or left and right in the binary tree), operate on it by recursively calling the template.

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

Abstraction over functions

I think you have seen, in your past, how functions provide abstraction over values:

```
(+ 32 (* 1 (/ 9 5)))    ;; num  
(+ 32 (* 22.5 (/ 9 5))) ;; num  
(+ 32 (* -31 (/ 9 5))) ;; num  
...  
  
;; num -> num  
(lambda (x) (+ 32 (* x (/ 9 5))))
```

a function taking an x saves us from having to write the same expression over and over again, with only one difference.

Abstraction over functions

```
; list-sum consumes a list of numbers and produces  
; the sum of those numbers.  
;  
; list num -> num  
(define/match (list-sum l)  
  [(['()) 0]  
  [((cons x xs)) (+ x (list-sum xs))])
```

```
; list-prod consumes a list of numbers and produces  
; the product of those numbers.  
;  
; list num -> num  
(define/match (list-prod l)  
  [(['()) 0]  
  [((cons x xs)) (* x (list-prod xs))])
```

Abstraction over functions

A function that consumes, as one of its arguments, another function is an example of a **higher-order function**.

```
; list-process consumes a binary operator on numbers
; and a list of numbers, and produces the result of
; applying that operator on the numbers.
;
; (num -> num -> num) -> list num -> num
(define (list-process f l)
  (match* (l)
    [(`() 0]
    [((cons x xs)) (f x (list-process f xs))]))

(define (list-sum l) (list-process + l))
(define (list-product l) (list-process * l))
```

Functions that produce values

```
;; num -> num
(define (add1 x) (+ 1 x))

;; num -> num
(define (add2 x) (+ 2 x))

...

;; num -> num
(define (add42 x) (+ 42 x)) ; ???

...
```

Functions that produce values

```
;; num -> (num -> num)  
(define (add-n n) (lambda (x) (+ x n)))
```


Functions that produce values

```
;; num -> (num -> num)
(define (add-n n) (lambda (x) (+ x n)))
```

```
;; num -> num
(define (add1 n) (add-n 1))
```

```
;; num -> num
(define (add2 n) (add-n 2))
```

Abstraction over functions

A function that produces a function when it is called is an example of a **higher-order function**.

```
;; num -> (num -> num)
(define (add-n n)
  (lambda (x) (+ x n)))

;; num -> num
(define (add1 n) (add-n 1))

;; num -> num
(define (add2 n) (add-n 2))
```

Higher order functions

Definition: A function f is said to be **higher-order** if:

- One or more of its arguments are a function
- The value it produces is a function

```
; list-process consumes a binary operator on numbers
; and a list of numbers, and produces the result of
; applying that operator on the numbers.
;
; (num -> num -> num) -> list num -> num
(define (list-process f l)
  (match* (l)
    [('()) 0]
    [((cons x xs)) (f x (list-process f xs))]))

(define (list-sum l) (list-process + l))
(define (list-product l) (list-process * l))
```

```
;; num -> (num -> num)
(define (add-n n)
  (lambda (x) (+ x n)))

;; num -> num
(define (add1 n) (add-n 1))

;; num -> num
(define (add2 n) (add-n 2))
```

Applying HOFs to lists

```
; num -> num
(define (add1 n) (+ 1 n))
(define (add2 n) (+ 2 n))
(define (add3 n) (+ 3 n))
; ...
(define (add42 n) (+ 42 n))
```

```
; listof num -> listof num
(define/match (add1-to-all l)
  (('()) '())
  (((cons n xs)) (cons (add1 n) (add1-to-all xs))))

(define/match (add2-to-all l)
  (('()) '())
  (((cons n xs)) (cons (add2 n) (add1-to-all xs))))

(define/match (add3-to-all l)
  (('()) '())
  (((cons n xs)) (cons (add3 n) (add1-to-all xs))))
```

Applying HOFs to lists

```
(define/match (function-to-all f l)
  ((_ '()) '())
  ((_ (cons n xs)) (cons (f n) (add1-to-all xs)))))
```

```
> (map add1 '(1 2 3))  
'(2 3 4)  
> (map (lambda (s) (string-append s ", world"))) '("hello" "howdy" "goodbye"))  
'("hello, world" "howdy, world" "goodbye, world")
```

map

- consumes a list of type A
- and a function that consumes a A and produces a B
- and returns a list of all the elements called with that function
- What is the return type of this function?
-

```
> (map add1 '(1 2 3))  
'(2 3 4)  
> (map (lambda (s) (string-append s ", world"))) '("hello" "howdy" "goodbye"))  
'("hello, world" "howdy, world" "goodbye, world")
```

map

- consumes a list of type A
- and a function that consumes a A and produces a B
- and returns a list of all the elements called with that function
- What is the return type of this function?

```
Prelude> :t map  
map :: (a -> b) -> [a] -> [b]  
Prelude> █
```

```
> (filter even? '(1 2 3 4 5))  
'(2 4)  
> (filter (λ (s) (< (string-length s) 4)) '("hello" "howdy" "hi"))  
'("hi")  
> |
```

filter

- consumes a list of type A
- and a predicate that consumes a A and produces a boolean
- and returns a list of all the elements that satisfy this predicate
- What is the return type of this function?


```
> (filter even? '(1 2 3 4 5))  
'(2 4)  
> (filter (λ (s) (< (string-length s) 4)) '("hello" "howdy" "hi"))  
'("hi")  
> |
```

filter

- consumes a list of type A
- and a predicate that consumes a A and produces a boolean
- and returns a list of all the elements that satisfy this predicate
- What is the return type of this function?

```
Prelude> :t filter  
filter :: (a -> Bool) -> [a] -> [a]  
Prelude> |
```

compose()

One final higher-order function before we break for the day.

What's the type signature for this function?

```
; ?? -> ?? -> ??  
(define (compose f g)  
  (lambda (x) (f (g x))))
```

compose()

```
; (? -> ?) -> (? -> ?) -> (? -> ?)
(define (compose f g)
  (lambda (x) (f (g x))))
```

compose()

```
; (b -> c) -> (a -> b) -> (a -> c)
(define (compose f g)
  (lambda (x) (f (g x))))
```

compose()

Proposal: Let's say we worked backwards and I asked you to write a function that satisfies this polymorphic type signature. I put it to you that the ONLY function that satisfies is it the implementation below.

```
; (b -> c) -> (a -> b) -> (a -> c)
(define (xyzzzy f g)
  ...)
```



we only have one choice!

Pause and ponder:

- Is this true of polymorphic map?
- Is this true of polymorphic filter?

```
; (b -> c) -> (a -> b) -> (a -> c)
(define (compose f g)
  (lambda (x) (f (g x))))
```



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
