CPSC 312

Functional and Logic Programming

November 3, 2015

Assignment 4

Your first Haskell assignment is coming up soon (in the next 48 hours).

From last session..

- We talked about some basic data types in Haskell
- We saw a variety of Haskell syntax and
- We implemented two functions in class, using the approach of abstraction: the quadratic formula and the loan payment formula.

The Loan Payment Formula

$$m = \frac{(P/1200)*(1+P/1200)^N *L}{(1+P/1200)^N -1}$$

```
monthlyPayment :: Float -> Float -> Int -> Float
monthlyPayment p l n = (f1 * f2 * l) / (f2-1)
 where
  f1 = p/1200
  f2 = (1+p/1200)^n
```

How is a query evaluated?

```
> monthlyPayment 20 10000 24
```

The Substitution Model of Evaluation

```
> monthlyPayment 20 10000 24
```

The behaviour of pure functional programs can be examined easily through *the substitution model of evaluation*.

Substitution model of evaluation

The substitution model of evaluation simply says that when the interpreter is given a function name with arguments to evaluate, it does so by retrieving the function body

```
> monthlyPayment 20 10000 24
```

```
monthlyPayment p l n = (f1 * f2 * l) / (f2-1)
 where
  f1 = p/1200
  f2 = (1+p/1200)^n
```

Substitution model of evaluation

The substitution model of evaluation simply says that when the interpreter is given a function name with arguments to evaluate, it does so by retrieving the function body

```
> monthlyPayment 20 10000 24
```

```
monthlyPayment p l n = (f1 * f2 * l) / (f2-1)
 where
  f1 = p/1200
  f2 = (1+p/1200)^n
```

Substitution model of evaluation

...and making the appropriate substitutions

```
> monthlyPayment 20 10000 24

monthlyPayment 20 10000 24 = (f1 * f2 * 10000) / (f2-
1)

 where

  f1 = 20/1200

  f2 = (1+20/1200)^24
```

Substitution model of evaluation

Similarly f1 and f2 in the definition of monthly payment are also replaced by their bodies:

```
> monthlyPayment 20 10000 24

monthlyPayment 20 10000 24 =
((20/1200) * (1+20/1200)^24 * 10000) /
((1+20/1200)^24 -1)
```

Substitution model of evaluation

Similarly f1 and f2 in the definition of monthly payment are also replaced by their bodies:

```
> monthlyPayment 20 10000 24

monthlyPayment 20 10000 24 = ((20/1200) *
(1+20/1200)^24 * 10000) / ((1+20/1200)^24
-1)
```

The original function invocation is replaced with this new definition to be evaluated.

Substitution model of evaluation

Similarly f1 and f2 in the definition of monthly payment are also replaced by their bodies:

```
> monthlyPayment 20 10000 24

monthlyPayment 20 10000 24 = ((20/1200) *
(1+20/1200)^24 * 10000) / ((1+20/1200)^24
-1)
```

The original function invocation is replaced with this new definition to be evaluated.

This is referential transparency in action.

Questions?

Iteration

How do you create a loop with Haskell?

Iteration

How do you create a loop with Haskell? You don't.

Loops are all about state changes and assignment statements, and you now abstain from all that inelegant stuff. And as an added bonus, there's no extra syntax that you have to remember for three or more different kinds of loops.

So let's ask that question another way: How do you get iterative behaviour from Haskell?

Iteration == recursion

Let's say you want to define a function that takes a non-negative integer n as a parameter and returns the sum of all integers from 1 through n. The nice thing about functional programming is that all you need to know is the conditional and the function call - there's no loop syntax.

Iteration == recursion

Let's say you want to define a function that takes a non-negative integer n as a parameter and returns the sum of all integers from 1 through n. The nice thing about functional programming is that all you need to know is the conditional and the function call - there's no loop syntax.

```
sumints :: Int -> Int
```

Iteration == recursion

Let's say you want to define a function that takes a non-negative integer n as a parameter and returns the sum of all integers from 1 through n. The nice thing about functional programming is that all you need to know is the conditional and the function call - there's no loop syntax.

```
sumints :: Int -> Int
sumints n
```

Iteration == recursion

Let's say you want to define a function that takes a non-negative integer n as a parameter and returns the sum of all integers from 1 through n. The nice thing about functional programming is that all you need to know is the conditional and the function call - there's no loop syntax.

```
sumints :: Int -> Int
sumints n
   | n == 1        = 1
```

Iteration == recursion

Let's say you want to define a function that takes a non-negative integer n as a parameter and returns the sum of all integers from 1 through n. The nice thing about functional programming is that all you need to know is the conditional and the function call - there's no loop syntax.

```
sumints :: Int -> Int
sumints n
    | n == 1        = 1
    | otherwise     = n + sumints (n - 1)
```

Recursion == induction

But let's take a step back and use induction:

```
sumints 1 =  1
```

Recursion == induction

But let's take a step back and use induction:

```
sumints 1 =  1
sumints 2 =  3 = 2 + 1 = 2 + sumints 1
```

Recursion == induction

But let's take a step back and use induction:

```
sumints 1 =  1
sumints 2 =  3 = 2 + 1 = 2 + sumints 1
sumints 3 =  6 = 3 + 3 = 3 + sumints 2
```

Recursion == induction

But let's take a step back and use induction:

```
sumints 1 =  1
sumints 2 =  3 = 2 + 1 = 2 + sumints 1
sumints 3 =  6 = 3 + 3 = 3 + sumints 2
sumints 4 = 10 = 4 + 6 = 4 + sumints 3
```

Recursion == induction

But let's take a step back and use induction:

```
sumints 1 =  1
sumints 2 =  3 = 2 + 1 = 2 + sumints 1
sumints 3 =  6 = 3 + 3 = 3 + sumints 2
sumints 4 = 10 = 4 + 6 = 4 + sumints 3
```

Can you see a pattern?

Recursion == induction

But let's take a step back and use induction:

```
sumints 1 =  1
sumints 2 =  3 = 2 + 1 = 2 + sumints 1
sumints 3 =  6 = 3 + 3 = 3 + sumints 2
sumints 4 = 10 = 4 + 6 = 4 + sumints 3
```

Can you see a pattern?

```
sumints n =              n + sumints (n - 1)
```

Recursion using abstraction

Another way to think about implementing a function recursively, is to think from the perspective of abstraction which we saw before. Ask:

If I was given the value of sumints (n-1), how could I calculate sumints n from it?

This is how:

```
sumints n = n + sumints (n - 1)
```

Return to recursion

A recursive procedure consists of three parts:

1. The base case or termination condition. Usually the first thing done upon entering a recursive procedure

2. The reduction step -- the operation that moves the computation closer to the termination condition

3. The recursive procedure call itself

Return to recursion

base case/termination

```
sumints :: Int -> Int
sumints n
    | n == 1        = 1
    | otherwise     = n + sumints (n - 1)
```

recursive call

reduction step

Return to recursion

OK, now let’s do factorial in Haskell...

Return to recursion

OK, now let's do factorial in Haskell...

```
factorial :: Int -> Int
```

Return to recursion

OK, now let's do factorial in Haskell...

```
factorial :: Int -> Int
factorial n
```

Return to recursion

OK, now let's do factorial in Haskell...

```
factorial :: Int -> Int
factorial n
   | n == 0         = 1
```

Return to recursion

OK, now let's do factorial in Haskell...

```
factorial :: Int -> Int
factorial n
    | n == 0          = 1
    | otherwise       = n * factorial (n - 1)
```

Return to recursion

OK, now let's do factorial in Haskell...

```
factorial :: Int -> Int
factorial n
    | n == 0          = 1
    | otherwise       = n * factorial (n - 1)
```

Here's what the substitution model of evaluation says about how factorial in Haskell works:

Return to recursion

OK, now do factorial in Haskell...

```
factorial :: Int -> Int
factorial n
    | n == 0          = 1
    | otherwise       = n * factorial (n - 1)

> factorial 3
```

Return to recursion

OK, now do factorial in Haskell...

```
factorial :: Int -> Int
factorial n
    | n == 0          = 1
    | otherwise       = n * factorial (n - 1)

> factorial 3
  3 * factorial 2
```

Return to recursion

OK, now do factorial in Haskell...

```
factorial :: Int -> Int
factorial n
   | n == 0        = 1
   | otherwise     = n * factorial (n - 1)

> factorial 3
  3 * factorial 2
  3 * 2 * factorial 1
```

Return to recursion

OK, now do factorial in Haskell...

```
factorial :: Int -> Int
factorial n
    | n == 0          = 1
    | otherwise       = n * factorial (n - 1)

> factorial 3
  3 * factorial 2
  3 * 2 * factorial 1
  3 * 2 * 1 * factorial 0
```

Return to recursion

OK, now do factorial in Haskell...

```
factorial :: Int -> Int
factorial n
   | n == 0         = 1
   | otherwise      = n * factorial (n - 1)

> factorial 3
  3 * factorial 2
  3 * 2 * factorial 1
  3 * 2 * 1 * factorial 0
  3 * 2 * 1 * 1
```

Return to recursion

OK, now do factorial in Haskell...

```
factorial :: Int -> Int
factorial n
    | n == 0          = 1
    | otherwise       = n * factorial (n - 1)

> factorial 3
  3 * factorial 2
  3 * 2 * factorial 1
  3 * 2 * 1 * factorial 0
  3 * 2 * 1 * 1
  3 * 2 * 1
```

Return to recursion

OK, now do factorial in Haskell...

```
factorial :: Int -> Int
factorial n
    | n == 0           = 1
    | otherwise        = n * factorial (n - 1)

> factorial 3
  3 * factorial 2
  3 * 2 * factorial 1
  3 * 2 * 1 * factorial 0
  3 * 2 * 1 * 1
  3 * 2 * 1
  3 * 2
```

Return to recursion

OK, now do factorial in Haskell...

```
factorial :: Int -> Int
factorial n
    | n == 0          = 1
    | otherwise       = n * factorial (n - 1)

> factorial 3
  3 * factorial 2
  3 * 2 * factorial 1
  3 * 2 * 1 * factorial 0
  3 * 2 * 1 * 1
  3 * 2 * 1
  3 * 2
  6
```

Return to recursion

OK, now do factorial in Haskell...

```
factorial :: Int -> Int
factorial n
    | n == 0        = 1
    | otherwise     = n * factorial (n - 1)

> factorial 3
  6
```

Return to recursion

Let's review what happens in terms of the behaviour of the activation stack:

```
fact n
   | n == 0      = 1
   | otherwise   =
      n * fact (n -
1)
```

```
> fact 3
```

Return to recursion

Let's review what happens in terms of the behaviour of the activation stack:

```
fact n
   | n == 0      = 1
   | otherwise   =
         n * fact (n -
1)
```

```
> fact 3
```

```
fact 3
```

Return to recursion

Let's review what happens in terms of the behaviour of the activation stack:

```
fact n
   | n == 0      = 1
   | otherwise   =
      n * fact (n -
1)
```

```
> fact 3
```

```
3 * fact 2
```

Return to recursion

Let's review what happens in terms of the behaviour of the activation stack:

```
fact n
   | n == 0      = 1
   | otherwise   =
         n * fact (n -
1)
```

```
> fact 3
```

```
fact 2

3 * fact 2
```

Return to recursion

Let's review what happens in terms of the behaviour of the activation stack:

```
fact n
   | n == 0      = 1
   | otherwise   =
         n * fact (n -
1)
```

```
> fact 3
```

```
2 * fact 1
3 * fact 2
```

Return to recursion

Let's review what happens in terms of the behaviour of the activation stack:

```
fact n
   | n == 0      = 1
   | otherwise   =
      n * fact (n -
1)
```

```
> fact 3
```

```
fact 1
2 * fact 1
3 * fact 2
```

Return to recursion

Let's review what happens in terms of the behaviour of the activation stack:

```
fact n
   | n == 0      = 1
   | otherwise   =
         n * fact (n -
1)
```

```
> fact 3
```

```
1 * fact 0

2 * fact 1

3 * fact 2
```

Return to recursion

Let's review what happens in terms of the behaviour of the activation stack:

```
fact n
   | n == 0      = 1
   | otherwise   =
      n * fact (n -
1)
```

```
> fact 3
```

```
fact 0
1 * fact 0
2 * fact 1
3 * fact 2
```

Return to recursion

Let's review what happens in terms of the behaviour of the activation stack:

```
fact n
   | n == 0      = 1
   | otherwise   =
      n * fact (n -
1)
```

```
> fact 3
```

```
1
1 * fact 0
2 * fact 1
3 * fact 2
```

Return to recursion

Let's review what happens in terms of the behaviour of the activation stack:

```
fact n
   | n == 0      = 1
   | otherwise   =
      n * fact (n -
1)
```

```
> fact 3
```

```
1 * 1
2 * fact 1
3 * fact 2
```

Return to recursion

Let's review what happens in terms of the behaviour of the activation stack:

```
fact n
   | n == 0      = 1
   | otherwise   =
         n * fact (n -
1)
```

```
> fact 3
```

```
1
2 * fact 1
3 * fact 2
```

Return to recursion

Let's review what happens in terms of the behaviour of the activation stack:

```
fact n
   | n == 0      = 1
   | otherwise   =
         n * fact (n -
1)
```

```
> fact 3
```

```
2 * 1
3 * fact 2
```

Return to recursion

Let's review what happens in terms of the behaviour of the activation stack:

```
fact n
   | n == 0      = 1
   | otherwise   =
      n * fact (n -
1)
```

```
> fact 3
```

```
2
3 * fact 2
```

Return to recursion

Let's review what happens in terms of the behaviour of the activation stack:

```
fact n
   | n == 0      = 1
   | otherwise   =
      n * fact (n -
1)
```

```
> fact 3
```

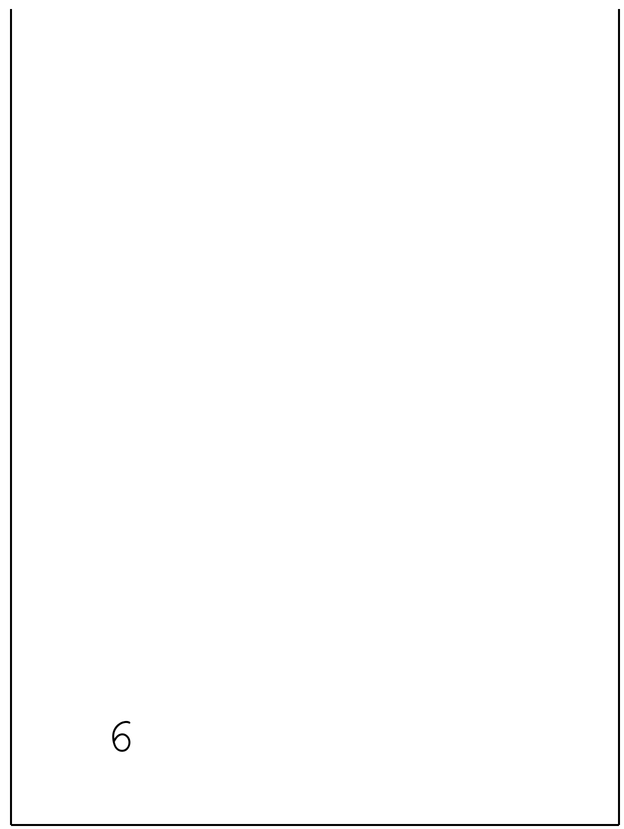
```
3 * 2
```

Return to recursion

Let's review what happens in terms of the behaviour of the activation stack:

```
fact n
   | n == 0      = 1
   | otherwise   =
      n * fact (n -
1)
```

```
> fact 3
```



Return to recursion

Let's review what happens in terms of the behaviour of the activation stack:

```
fact n
   | n == 0      = 1
   | otherwise   =
         n * fact (n -
1)
```

```
> fact 3
6

>
```

Return to recursion

Note 1: this is really simplified and high-level...there are really many more details being taken care of

```
fact n
   | n == 0      = 1
   | otherwise   =
         n * fact (n -
1)
```

```
> fact 3
6

>
```

Return to recursion

Note 2: it's also a bit of a lie...some language processors would optimize this recursion into a loop

```
fact n
   | n == 0      = 1
   | otherwise   =
         n * fact (n -
1)
```

```
> fact 3
6

>
```

Return to recursion

Note 3: while this, sort of, accurately describes what happens in Java, or with some C++ compilers, or Scheme/Racket, it doesn't really depict what Haskell does (more on this later). It's still a nice visualization to help you understand how, in general, recursion is implemented in many programming languages, so that's ok.

The Problem with Recursion

A common complaint about recursion, as we've just seen, is the resource consumption in terms of memory (activation stack space) as well as time (handling the function calls and putting frames on/taking frames off the stack).

The culprit here is the repeated postponement of computations by pushing those computations on the stack.

The Problem with Recursion

But what if there were a type of recursion that worked without postponing computations? If this were so, we could have recursion using O(1) (i.e., constant, regardless of the size of the input n) stack space instead of O(n) (i.e., increasing linearly in proportion to the size of the input n) stack space.

The Problem with Recursion

But what if there were a type of recursion that worked without postponing computations? If this were so, we could have recursion using O(1) (i.e., constant, regardless of the size of the input n) stack space instead of O(n) (i.e., increasing linearly in proportion to the size of the input n) stack space.

This type of recursion exists, and it's inspired by the observation that computations looking like this:

```
n * factorial (n - 1)
```

can also be represented this way:

```
n * (n - 1) * (n - 2) * ... * 1
```

The Problem with Recursion

For example, factorial 4 could be computed like this:

```
4 * 3 * 2 * 1
```

The Problem with Recursion

For example, factorial 4 could be computed like this:

```
4 * 3 * 2 * 1  becomes

12 * 2 * 1
```

The Problem with Recursion

For example, factorial 4 could be computed like this:

```
4 * 3 * 2 * 1  becomes

12 * 2 * 1     becomes

24 * 1
```

The Problem with Recursion

For example, factorial 4 could be computed like this:

```
4 * 3 * 2 * 1  becomes

12 * 2 * 1     becomes

24 * 1         becomes

24
```

The Problem with Recursion

In other words, I could start with an accumulator "variable" to hold the product,

product:

The Problem with Recursion

In other words, I could start with an accumulator "variable" to hold the product, initialize it to 1,

product:	1

The Problem with Recursion

In other words, I could start with an accumulator "variable" to hold the product, initialize it to 1, multiply it by 4,

product: 4

The Problem with Recursion

In other words, I could start with an accumulator "variable" to hold the product, initialize it to 1, multiply it by 4, then multiply that value by 3,

product: 12

The Problem with Recursion

In other words, I could start with an accumulator "variable" to hold the product, initialize it to 1, multiply it by 4, then multiply that value by 3, and then multiply that value by 2,

product:	24

The Problem with Recursion

In other words, I could start with an accumulator "variable" to hold the product, initialize it to 1, multiply it by 4, then multiply that value by 3, and then multiply that value by 2, and finally multiply that value by 1 to give the result of the function call `factorial 4`:

product: 24

The Problem with Recursion

In other words, I could start with an accumulator "variable" to hold the product, initialize it to 1, multiply it by 4, then multiply that value by 3, and then multiply that value by 2, and finally multiply that value by 1 to give the result of the function call `factorial 4`:

How do we make this happen?

product:	24

Tail recursion

This type of recursion is called tail recursion, and it usually involves the introduction of an additional "variable" to hold the partially-computed result instead of storing postponed computations on the stack. It might be easier just to see an example and then talk about it.

Tail recursion

```
fact_tr :: Int -> Int
fact_tr n = fact_tr_helper n 1

fact_tr_helper :: Int -> Int -> Int
fact_tr_helper n product
    | n == 0        = product
    | otherwise     = fact_tr_helper (n - 1) (product * n)
```

Tail recursion

```
fact_tr :: Int -> Int
fact_tr n = fact_tr_helper n 1

fact_tr_helper :: Int -> Int -> Int
fact_tr_helper n product
    | n == 0          = product
    | otherwise       = fact_tr_helper (n - 1) (product * n)

> fact_tr 4
```

Tail recursion

```
fact_tr :: Int -> Int
fact_tr n = fact_tr_helper n 1

fact_tr_helper :: Int -> Int -> Int
fact_tr_helper n product
    | n == 0         = product
    | otherwise      = fact_tr_helper (n - 1) (product * n)

> fact_tr 4
  fact_tr_helper 4 1
```

Tail recursion

```
fact_tr :: Int -> Int
fact_tr n = fact_tr_helper n 1

fact_tr_helper :: Int -> Int -> Int
fact_tr_helper n product
    | n == 0         = product
    | otherwise      = fact_tr_helper (n - 1) (product * n)

> fact_tr 4
  fact_tr_helper 4 1
  fact_tr_helper 3 4
```

Tail recursion

```
fact_tr :: Int -> Int
fact_tr n = fact_tr_helper n 1

fact_tr_helper :: Int -> Int -> Int
fact_tr_helper n product
    | n == 0        = product
    | otherwise     = fact_tr_helper (n - 1) (product * n)

> fact_tr 4
  fact_tr_helper 4 1
  fact_tr_helper 3 4
  fact_tr_helper 2 12
```

Tail recursion

```
fact_tr :: Int -> Int
fact_tr n = fact_tr_helper n 1

fact_tr_helper :: Int -> Int -> Int
fact_tr_helper n product
   | n == 0         = product
   | otherwise      = fact_tr_helper (n - 1) (product * n)

> fact_tr 4
  fact_tr_helper 4 1
  fact_tr_helper 3 4
  fact_tr_helper 2 12
  fact_tr_helper 1 24
```

Tail recursion

```
fact_tr :: Int -> Int
fact_tr n = fact_tr_helper n 1

fact_tr_helper :: Int -> Int -> Int
fact_tr_helper n product
   | n == 0        = product
   | otherwise     = fact_tr_helper (n - 1) (product * n)

> fact_tr 4
  fact_tr_helper 4 1
  fact_tr_helper 3 4
  fact_tr_helper 2 12
  fact_tr_helper 1 24
  fact_tr_helper 0 24
```

Tail recursion

```
fact_tr :: Int -> Int
fact_tr n = fact_tr_helper n 1

fact_tr_helper :: Int -> Int -> Int
fact_tr_helper n product
   | n == 0          = product
   | otherwise       = fact_tr_helper (n - 1) (product * n)

> fact_tr 4
  fact_tr_helper 4 1
  fact_tr_helper 3 4
  fact_tr_helper 2 12
  fact_tr_helper 1 24
  fact_tr_helper 0 24
  24
```

Tail recursion

```
fact_tr :: Int -> Int
fact_tr n = fact_tr_helper n 1

fact_tr_helper :: Int -> Int -> Int
fact_tr_helper n product
    | n == 0         = product
    | otherwise      = fact_tr_helper (n - 1) (product * n)

> fact_tr 4
  fact_tr_helper 4 1
  fact_tr_helper 3 4
  fact_tr_helper 2 12
  fact_tr_helper 1 24
  fact_tr_helper 0 24
  24
```

No postponed computations! Thus no linearly-increasing stack usage...

Tail recursion

```
fact_tr :: Int -> Int
fact_tr n = fact_tr_helper n 1

fact_tr_helper :: Int -> Int -> Int
fact_tr_helper n product
    | n == 0        = product
    | otherwise     = fact_tr_helper (n - 1) (product * n)

> fact_tr 4
  24
```

Tail recursion

In theory, it works like this:

```
fact_tr n =
   fact_tr_helper n 1

fact_tr_helper n product
   | n == 0
     = product
   | otherwise
     = fact_tr_helper
         (n - 1)
         (product * n)
```

```
> fact_tr 4
```

Tail recursion

In theory, it works like this:

```
fact_tr n =
   fact_tr_helper n 1

fact_tr_helper n product
   | n == 0
     = product
   | otherwise
     = fact_tr_helper
         (n - 1)
         (product * n)
```

```
> fact_tr 4
```

```
fact_tr 4
```

Tail recursion

In theory, it works like this:

```
fact_tr n =
   fact_tr_helper n 1

fact_tr_helper n product
   | n == 0
     = product
   | otherwise
     = fact_tr_helper
         (n - 1)
         (product * n)
```

```
> fact_tr 4
```

```
fact_tr_helper 4 1
```

Tail recursion

In theory, it works like this:

```
fact_tr n =
   fact_tr_helper n 1

fact_tr_helper n product
   | n == 0
     = product
   | otherwise
     = fact_tr_helper
         (n - 1)
         (product * n)
```

```
> fact_tr 4
```

```
fact_tr_helper 3 4
```

Tail recursion

In theory, it works like this:

```
fact_tr n =
   fact_tr_helper n 1

fact_tr_helper n product
   | n == 0
     = product
   | otherwise
     = fact_tr_helper
          (n - 1)
          (product * n)
```

```
> fact_tr 4
```

```
fact_tr_helper 2 12
```

Tail recursion

In theory, it works like this:

```
fact_tr n =
   fact_tr_helper n 1

fact_tr_helper n product
   | n == 0
     = product
   | otherwise
     = fact_tr_helper
         (n - 1)
         (product * n)
```

```
> fact_tr 4
```

```
fact_tr_helper 1 24
```

Tail recursion

In theory, it works like this:

```
fact_tr n =
   fact_tr_helper n 1

fact_tr_helper n product
   | n == 0
     = product
   | otherwise
     = fact_tr_helper
         (n - 1)
         (product * n)
```

```
> fact_tr 4
```

```
fact_tr_helper 0 24
```

Tail recursion

In theory, it works like this:

```
fact_tr n =
   fact_tr_helper n 1

fact_tr_helper n product
   | n == 0
     = product
   | otherwise
     = fact_tr_helper
          (n - 1)
          (product * n)
```

```
> fact_tr 4
```

```
24
```

Tail recursion

In theory, it works like this:

```
fact_tr n =
   fact_tr_helper n 1

fact_tr_helper n product
   | n == 0
     = product
   | otherwise
     = fact_tr_helper
          (n - 1)
          (product * n)
```

```
> fact_tr 4
24

>
```

Tail recursion

In theory, it works like this, but in practice it might not work like this at all.

```
fact_tr n =
   fact_tr_helper n 1

fact_tr_helper n product
   | n == 0
     = product
   | otherwise
     = fact_tr_helper
         (n - 1)
         (product * n)
```

```
> fact_tr 4
24

>
```

Tail recursion

In theory, it works like this, but in practice it might not work like this at all.

Not all programming languages can take advantage of tail recursion in this way. It's implementation dependent.

Scheme, by definition, must work this way. Many other functional languages will do this. Java will optimize tail recursion, as will languages built on the Java Virtual Machine (Scala, Clojure). The GCC compiler does as well.

Tail recursion

Haskell can work this way, but by default it doesn't. Haskell employs **lazy evaluation** so even tail recursive calls are postponed (or *thunked*), which means we lose the tail recursion benefit you see in Racket. (It's ok, we get other benefits from lazy evaluation. We'll put that discussion in a thunk for later...)

Tail recursion

Compare – What's the big difference between these two?

```
fact_tr :: Int -> Int
fact_tr n = fact_tr_helper n 1

fact_tr_helper :: Int -> Int -> Int
fact_tr_helper n product
    | n == 0          = product
    | otherwise       = fact_tr_helper (n - 1) (product * n)
```

versus

```
factorial :: Int -> Int
factorial n
    | n == 0          = 1
    | otherwise       = n * factorial (n - 1)
```

Tail recursion

Some cautionary words:

"[T]he efficacy of tail recursion relies on the implementation. Moral: Functional programming languages are meant to be tools for rapid prototyping. Users should not spend too much time 'improving' their programs by hand while probably making them more obscure and possibly less efficient. Many such improvements can be made automatically or semi-automatically by the compiling system which knows which ones really are improvements."

-- A.J.T. Davie in
An Introduction to Functional Programming Systems Using Haskell

Tail recursion

Our moral: For now, don't use tail recursion because you think it will get you a more efficient Haskell program. Do use tail recursion if it makes more sense to you than ordinary vanilla recursion (more formally known in some circles as *augmenting recursion* or *natural recursion*).

Recursion: so misunderstood (sigh)

"Recursion isn't useful very often, but when used judiciously it produces exceptionally elegant solutions.... In general, recursion leads to small code and slow execution and chews up stack space. For a small group of problems, recursion can produce simple, elegant solutions. For a slightly larger group of problems, it can produce simple, elegant, hard-to-understand solutions. For most problems, it produces massively complicated solutions -- in those cases, simple iteration is usually more understandable. Use recursion selectively."

Steve McConnell in Code Complete

Recursion: so misunderstood (sigh)

"Recursion isn't useful very often, but when used judiciously it produces exceptionally elegant solutions.... In general, recursion leads to small code and slow execution and chews up stack space."

As you've just seen, optimizing compilers can make this problem go away. If you don't have an optimizing compiler, tail recursion can help.

Recursion: so misunderstood (sigh)

More from McConnell:

“If a programmer who worked for me used recursion to compute a factorial, I'd hire someone else. Here's the recursive version of the factorial routine...” (in Pascal)

```
Function Factorial( Number: integer ): integer;
begin
   if ( Number = 1 ) then
      Factorial := 1
   else
      Factorial := Number * Factorial( Number - 1);
end;
```

Recursion: so misunderstood (sigh)

More from McConnell:

“In addition to being slow and making the use of run-time memory unpredictable, the recursive version of this routine is harder to understand than the iterative version. Here's the iterative version:”

```
Function Factorial( Number: integer ): integer;
var
   IntermediateResult: integer;
   Factor:             integer;
begin
   IntermediateResult := 1;
   for Factor := 2 to Number do
      IntermediateResult := IntermediateResult * Factor;
   Factorial := IntermediateResult;
end;
```

Recursion: so misunderstood (sigh)

Doesn't this just look like the mathematical definition of factorial?

```
Function Factorial( Number: integer ): integer;
begin
   if ( Number = 1 ) then
      Factorial := 1
   else
      Factorial := Number * Factorial( Number - 1);
end;
```

$$n! = \begin{cases} 1 & \text{if } n=0 \\ (n-1)! \times n & \text{if } n>0 \end{cases}$$

Recursion: so misunderstood (sigh)

This, on the other hand, is a good example of how imperative programming pushes algorithm implementation to follow the architecture:

```
Function Factorial( Number: integer ): integer;
var
   IntermediateResult: integer;
   Factor:             integer;
begin
   IntermediateResult := 1;
   for Factor := 2 to Number do
      IntermediateResult := IntermediateResult * Factor;
   Factorial := IntermediateResult;
end;
```

Recursion: so misunderstood (sigh)

Things to consider:

1. You now know that the recursive version isn't necessarily slow and doesn't necessarily chew up memory.

2. You also know that there may be reasons for employing recursion that are more important than efficiency. Think back to the "Most important open problem in programming languages -- programmer productivity" perspective. It's no less valid than McConnell's.

3. Understandability, like beauty, is in the eye of the beholder.

Questions?

Exercise

1. (Exercise 4.20, p. 86): Write a <u>recursive</u> function that computes the integer square of a given number n.

The integer square root of a positive integer n is the largest integer whose square is less than or equal to n.

The integer square root of 15 and 16 are 3 and 4, respectively.

2. (Exercise 2, p. 85) Implement the function `sumFacs` and `Integer->Integer`, such that

```
sumFacs n = fac 0 + fac 1 + … + fac (n-1) + fac n
```

Exercise

3. `sumFacs n = fac 0 + fac 1 + … + fac (n-1) + fac n`

Can you write a more general version of `sumFacs` that works for any given function? In this new function, `sumFacs` will be the equivalent of

```
sumFunction fac n
```

(Remember that functions are first-class values and can be passed as arguments to other functions!)

See page 85 for solution

Lazy Evaluation

Many (most?) languages employ strict evaluation. The arguments to a function are evaluated before the function is applied to the arguments.

Haskell doesn't evaluate arguments, or any expressions, until it has to. Instead, Haskell creates a "promise" to compute the expression. The record in memory that's used to keep track of an unevaluated expression is called a *thunk*.

thunk: an expression, frozen together with its environment (i.e. variable values), for later evaluation if and when needed. (similar to a closure)

Lazy Evaluation

So a function call in Haskell doesn't create a new frame on the call stack; it creates a thunk and defers evaluation until the value of the expression is needed. If the result is never used, the value will never be computed. That's lazy (or nonstrict) evaluation.

Lazy Evaluation

So a function call in Haskell doesn’t create a new frame on the call stack; it creates a thunk and defers evaluation until the value of the expression is needed. If the result is never used, the value will never be computed. That’s lazy (or nonstrict) evaluation.

When do thunks get evaluated? When it’s absolutely necessary. For example, if a conditional needs a value to go forward, the corresponding thunk(s) will be evaluated. If a value has to be printed on the console, the corresponding thunk(s) will be evaluated. And so on.

Lazy Evaluation

Remember this:

```
> monthlyPayment 20 10000 24

monthlyPayment 20 10000 24 = ((20/1200) *
(1+20/1200)^24 * 10000) / ((1+20/1200)^24
-1)
```

“The original function invocation is replaced with this new definition to be evaluated.”

This new definition is a thunk.

Lazy Evaluation

```
> monthlyPayment 20 10000 24

monthlyPayment 20 10000 24 = ((20/1200) *
(1+20/1200)^24 * 10000) / ((1+20/1200)^24
-1)
```

"The original function invocation is replaced with this new definition to be evaluated."

This new definition is a thunk.
The only reason the result evaluates when you type this into your interpreter is because the interpreter puts a "print" call before the call to function and that forces a "printable value" to be produced. An unevaluated thunk is not printable.

Lazy Evaluation

What if we have a real need to control space and time in Haskell? Can we force strict evaluation? Yes, there are ways, but those are techniques you can learn on your own if/when you decide to be a killer Haskell programmer. We won't need strict evaluation in CPSC 312.

Still, you should be able to read and write tail-recursive Haskell functions, if only because it's a style that some programmers find more understandable than "vanilla" recursion.

Multiple Recursion

We've only looked at simple recursions like this so far:

```
factorial :: Int -> Int
factorial n
  | n == 0        = 1
  | otherwise   = n * factorial (n - 1)
```

The next example is more complicated. Some functions will substitute multiple recursive procedure calls for the original procedure call.

Multiple Recursion

Here's how the Fibonacci numbers are computed:

fib(0) = 0

fib(1) = 1

fib(n) = fib(n - 1) + fib(n - 2)

How do you turn that into Haskell?

Multiple Recursion

fib(0) = 0

fib(1) = 1

fib(n) = fib(n - 1) + fib(n - 2)

```
fibonacci :: Int -> Int
fibonacci n
  | n == 0  = 0
  | n == 1  = 1
  | otherwise = fibonacci (n - 1) + fibonacci (n -
2)
```

Multiple Recursion

What happens when you run this with

n = 5? n = 10? n = 30? n = 40?

```
fibonacci :: Int -> Int
fibonacci n
  | n == 0  = 0
  | n == 1  = 1
  | otherwise = fibonacci (n - 1) + fibonacci (n -
2)
```

Why does that happen?

Multiple Recursion

Is there a solution that doesn't eat up stack space and time? Sure, we just have to find a solution that doesn't involve multiple recursion.

Multiple Recursion

Here's one way (and only one way):

```
fibfast :: Int -> Int
fibfast n
    | n < 2      = n
    | otherwise = fibup n 2 1 0
fibup :: Int -> Int -> Int -> Int -> Int
fibup max count nminus1 nminus2
    | max == count   = nminus1 + nminus2
    | otherwise    = fibup max (count + 1) (nminus1 +
nminus2) nminus1
```

Multiple Recursion

Here's one way (and only one way):

```
fibfast :: Int -> Int
fibfast n
   | n < 2      = n
   | otherwise = fibup n 2 1 0
fibup :: Int -> Int -> Int -> Int -> Int
fibup max count nminus1 nminus2
   | max == count    = nminus1 + nminus2
   | otherwise    = fibup max (count + 1) (nminus1 +
nminus2) nminus1
```

Not necessarily easy to figure out at first, but after you look at it for awhile you say "oh, that makes sense!"

Questions?