CS 320: Concepts of Programming Languages

Wayne Snyder
Computer Science Department
Boston University

Lecture 06: Programming with Functions

- Functions as First-Class Values
- Examples of functional programming: Map and Filter
- Lambda Expressions
- Functions on functions
- Modules

Reading: Hutton Ch. 4, beginning of Ch. 7

Programming with Functions

In functional programming, we want to treat functions as "first-class values," i.e., having the same "rights" as any other kind of data, i.e, functions, like data, can be

- passed as parameters
- stored in data structures
- represented as values without having to assign to a name.
- manipulated by other functions to create new functions

In most programming languages, functions are not treated in this way, but we will find that in Haskell this is pursued to the greatest extend possible.

This opens up a world of possibilities for algorithms that are not possible in other languages; often these algorithms are more concise and elegant than in other languages. Of course this is a matter of taste! We will at least explore this possibility, and add to your toolkit of possibilities for programming, and you make up your mind after the course is over!

Functional Programming Paradigms

Let us first consider what it would mean to allow functions to be passed as parameters... suppose we wanted to increment every member of an Integer list:

```
incr :: Integer -> Integer
incr x = x + 1

incrList :: [Integer] -> [Integer]
incrList []     = []
incrList (x:xs) = (incr x):(incrList xs)
```

```
Main> incrList [3,4]
[4,5]
```

Functional Programming Paradigms

- passed as parameters
- stored in data structures
- represented as values without having to assign to a name.
- manipulated by other functions

Then later we want to test every member of a list to see if it is even:

```
isEven :: Integer -> Bool
isEven x = x `mod` 2 == 0

isEvenList :: [Integer] -> [Bool]
isEvenList []     = []
isEvenList (x:xs) = (isEven x):(isEvenList xs)
```

```
Main> isEvenList [3,4]
[False,True]
```

Functional Programming Paradigms

- passed as parameters
- stored in data structures
- represented as values without having to assign to a name.
- manipulated by other functions

Hm... these look similar:

```
incr :: Integer -> Integer
incr x = x + 1

incrList :: [Integer] -> [Integer]
incrList []     = []
incrList (x:xs) = (incr x):(incrList xs)

isEven :: Integer -> Bool
isEven x = x `mod` 2 == 0

isEvenList :: [Integer] -> [Bool]
isEvenList []     = []
isEvenList (x:xs) = (isEven x):(isEvenList xs)
```

What to do? Clearly, we should write a function that keeps the common elements and **abstracts** out the differences using parameters/variables.

Functional Programming Paradigms

- passed as parameters
- stored in data structures
- represented as values without having to assign to a name.
- manipulated by other functions

But to abstract out the common core of this paradigm, and make parameters of the differences, we have to

- Parameterize the types using polymorphism and type variables
- Parameterize the function by allowing a function to be passed as a parameter.

```
isEven :: Integer -> Bool
isEven x = x `mod` 2 == 0

isEvenList :: [Integer] -> [Bool]
isEvenList [] = []
isEvenList (x:xs) = (isEven x):(isEvenList xs)
```

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = (f x):(map f xs)
```

```
isEvenList = map isEven
```

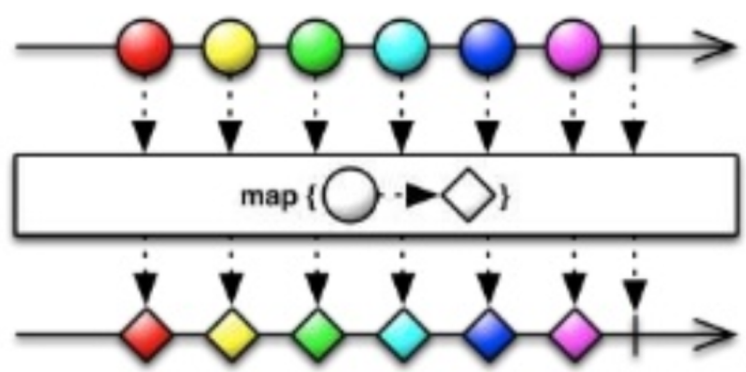
Functional Programming Paradigms

- passed as parameters
- stored in data structures
- represented as values without having to assign to a name.
- manipulated by other functions

Map is a common function and is defined in the Prelude (with built-in lists):

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x):(map f xs)
```

```
Main> map incr [3,5]
[4,6]

Main> map times2 [3,5]
[6,10]
```



[3]

Functional Programming Paradigms

- passed as parameters
- stored in data structures
- represented as values without having to assign to a name.
- manipulated by other functions

Ok, here is another common paradigm: filter a list by only allowing elements that satisfy some predicate (Boolean test):

```
isEven :: Integer -> Bool
isEven x = x `mod` 2 == 0

filterEvenList :: [Integer] -> [Integer]
filterEvenList []                    = []
filterEvenList (x:xs) | isEven x     = x:(filterEvenList xs)
                      | otherwise = filterEvenList xs
```

```
Main> filterEvenList [2,3,4]
[2,4]
```

Functional Programming Paradigms

- passed as parameters
- stored in data structures
- represented as values without having to assign to a name.
- manipulated by other functions

We abstract out the common core of this algorithm to obtain another common function defined in the Prelude:

```
isEven :: Integer -> Bool
isEven x = x `mod` 2 == 0

filterEvenList :: [Integer] -> [Integer]
filterEvenList []                      = []
filterEvenList (x:xs) | isEven x   = x:(filterEvenList xs)
                      | otherwise = filterEvenList xs
```

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []                   = []
filter p (x:xs) | p x         = x:(filter p xs)
                | otherwise = filter p xs
```

```
Main> filter isEven [2,3,4]
[2,4]
```

Functional Programming Paradigms

- passed as parameters
- stored in data structures
- represented as values without having to assign to a name.
- manipulated by other functions

So we have demonstrated the first in our list of desirable features for functional programming: Functions can be

- passed as parameters
- stored in data structures
- manipulated by other functions
- represented as values without having to assign to a name.

How about storing in data structures? **No problem in Haskell!**

Suppose we want to apply a list of functions to a list of values?

```
[incr,times2,decr]

[4,5,9]

[5,10,8]
```

```haskell
incr :: Integer -> Integer
incr x = x + 1

decr :: Integer -> Integer
decr x = x - 1

times2 :: Integer -> Integer
times2 x = x * 2
```

Functional Programming Paradigms

- passed as parameters
- stored in data structures
- represented as values without having to assign to a name.
- manipulated by other functions

This is not a standard Prelude function, but easy to write!
Of course it should be polymorphic:

```
applyList :: [a -> b] -> [a] -> [b]
applyList [] _
applyList _ []
applyList (f:fs) (x:xs) = (f x):(applyList fs xs)
```

```
Main> funcList = [incr,times2,decr]

Main> argList = [4,5,9]
Main> applyList funcList argList
[5,10,8]
```

```
incr :: Integer -> Integer
incr x = x + 1

decr :: Integer -> Integer
decr x = x - 1

times2 :: Integer -> Integer
times2 x = x * 2
```

Functional Programming Paradigms

- passed as parameters
- stored in data structures
- represented as values without having to assign to a name.
- manipulated by other functions

And then there is nothing to prevent us from manipulating functions like we would any other "value" that gets stored in a data structure:

```
Main> funcList = [incr,times2,decr]

Main> argList = [4,5,9]

Main> f = head funcList
Main> f 8
9

Main> applyList (tail funcList) (tail argList)
[10,8]

Main> (head (tail funcList)) (last argList)
18
```

This is just a consequence of referential transparency: the meaning of an expression is unchanged if we replace a subexpression by an equivalent subexpression.

Lambda Expressions in Haskell

- passed as parameters
- stored in data structures
- represented as values without having to assign to a name.
- manipulated by other functions

Ok, onward! How do we deal with the "value" of a function separate from a identifier bound to a value?

```
   3     [5]    'a'         "Hi there"
```

```
Main> x = 3
```

```
Main> lst =  [5]
```

Ordinary data values don't HAVE to have a name: they exist separately from names, and are bound to a name when necessary. This is absolutely necessary during ordinary programming: we pass values to functions without having to name them (unless they enter the function):

```
Main>  incr  4
5
```

Can we treat functions the same way? Well, in Haskell, of course you can.... (also in Python)....

Lambda Expressions in Haskell

- passed as parameters
- stored in data structures
- represented as values without having to assign to a name.
- manipulated by other functions

Haskell allows you to write lambda expressions to represent the computational content of a function separate from its name. What's left? The list of parameters and the body of the function! These are sometimes called anonymous functions, but the term **lambda expression** is standard:

```
\<parameter> -> <body of function>
```

```
Main> f = \x -> x + 1
Main> f 4
5
Main> (\x y z -> x + y*z)  3  4 5
23
Main> (\x -> x + ( (\y -> y * 2) 6 ) )  10
22
```

Perhaps you have seen this in Python:

script.py IPython Shell

```
# Program to show the use of lambda functions

double = lambda x: x * 2

# Output: 10
print(double(5))
```

Or math notation:

$\lambda x.\ x+1$

Lambda Expressions in Haskell

One very useful feature of Haskell lambda expressions is that you can use patterns as the “bound variable,” but you have to watch out for non-exhaustive patterns, which will cause a warning!

```
Main> (\(x,y) -> x + y) (3,4)
7
Main> (\(x:xs) -> 2*x) [2,3,4]

<interactive>:135:2: warning: [-Wincomplete-uni-patterns]
    Pattern match(es) are non-exhaustive
    In a lambda abstraction: Patterns not matched: []
4
```

If you want to do multiple cases in a lambda, you'll have to use a case statement:

```
describe :: [a] -> String
describe = \xs -> case xs of
                    []  -> "empty"
                    [x] -> "singleton"
                    _   -> ″big!"
```

```
describe :: [a] -> String
describe []  = "empty"
describe [x] = "singleton"
describe _   = "big!"
```

Lambda Expressions in Haskell

One of the main uses of such anonymous functions is to avoid the use of separately-defined "helper functions" in functions such as map and filter:

```
Main> map (\x -> x + 1) [2,3,4]
[3,4,5]

Main> filter (\x -> x `mod` 2 == 0) [2,3]
[2]
```

or in any place where the name of a function is not really the point:

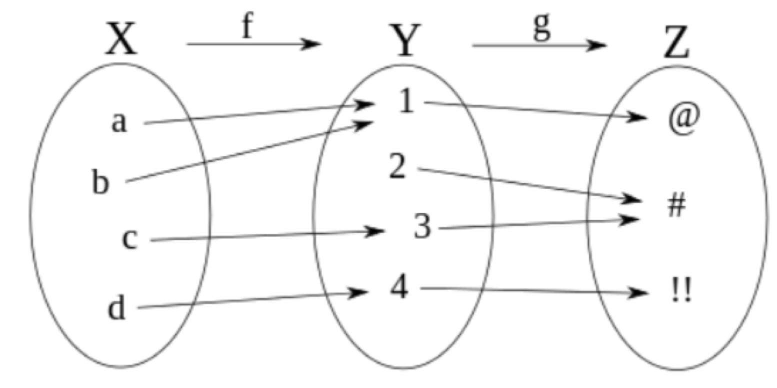
```
Main> funcList = [(\x -> x + 1),(\z -> z * 2)]

Main> applyList funcList  [2,5]
[3,10]
```

Higher-order Programming Paradigms

Reading: Hutton Ch. 7.5

Functions can be manipulated by other functions/operators to create new functions. In mathematics the most common such operator is function composition:

```
f ∘ g (x)  =  f(g(x))
```



Function composition in Haskell:

```
incr x = x + 1
times2 x = x * 2

plus1times2 = times2 . incr

Main> incr 2
3
Main> times2 3
6
Main> plus1times2 2
6
```

Function composition operator in Haskell is the period.

Higher-order Programming Paradigms

Reading: Hutton Ch. 7.5

There are many other functions which manipulate functions in useful ways… Here are a couple of my favorites!

```
-- exchange the order of arguments
--    for a binary function

flip :: (a -> b -> c) -> (b -> a -> c)
flip f = \y x -> f x y
```

```
Main> exp = flip (^)
Main> exp 2 3
9
```

Higher-order Programming Paradigms

Reading: Hutton Ch. 7.5

Function slices allow you to apply a binary infix function to one argument, leaving the other as a parameter:

```
Main> times2 = \x -> x * 2
Main> times2 4
8
Main> times3 = (*3)
Main> times3 4
12
Main> (*2) ((1+) 6)
14
Main> add0 = (`append` 0)
Main> add0 [2,4,0]


Main> map (`div` 2) [5,3]
[2,1]
```

Beta Reduction and Let Expressions

Recall: a lambda expression represents an anonymous function:

```
makePair :: a -> b -> (a,b)
makePair x y = (x,y)

makePair x = \y -> (x,y)

makePair = \x -> \y -> (x,y)

Main> makePair 3 True
(3,True)
```

By referential transparency, we can simply use the lambda expression and apply it directly to arguments:

```
Main> (\x -> \y -> (x,y)) 3 True
(3,True)
```

Beta Reduction and Let Expressions

We will study this much more in a few weeks, when we start to think about how to implement functional languages, but for now, we just define the concept of Beta-Reduction, which is simply substituting an argument for its parameter:

```
((\x -> <expression>) <argument>)
```

`=>  <expression>` with `x` replaced by `<argument>`

Examples:

```
Main> (\x -> (x,x))  4
(4,4)

Main>(\x -> [3,x,9]) 4
[3,4,9]

Main>(\x -> Just x) "hi"
Just "hi"

Main>(\x -> 5) 6
5
```

```
Main> (\x -> (\y -> (x,y))) 5 True
(5,True)

Main>(\x y -> [3,x,y]) 4 9
[3,4,9]

Main>(\x y -> \z -> [x,y,z]) 2 4 9
[2,4,9]

Main> (\x -> (\x -> (x,x))) 5 True
??
```

Beta Reduction and Lambda Expressions

We will study this much more in a few weeks, when we start to think about how to implement functional languages, but for now, we just define the concept of Beta-Reduction, which is simply substituting an argument for its parameter:

```
((\x -> <expression>) <argument>)
```

`=>  <expression>` with `x` replaced by `<argument>`

Examples:

```
Main> (\x -> (x,x))  4
(4,4)
Main>(\x -> [3,x,9]) 4
[3,4,9]
Main>(\x -> Just x) "hi"
Just "hi"
Main>(\x -> 5) 6
5
```

```
Main> (\x -> (\y -> (x,y))) 5 True
(5,True)
Main>(\x y -> [3,x,y]) 4 9
[3,4,9]
Main>(\x y -> \z -> [x,y,z]) 2 4 9
[2,4,9]
Main> (\x -> (\x -> (x,x))) 5 True
(True,True)
```

```
Why??
```

Scope in Haskell

The scope of a variable (e.g., local variable, parameter) is the region of the program where it is legal to refer to that variable.

```
Main> x

<interactive>:14:1: error: Variable not in scope: x
Main>
Main> x = 4
Main> x
4
```

In Java there are several kinds of scoping rules.....

Digression: Scope in Java

The scope of a variable (e.g., local variable, parameter) is the region of the program where it is legal to refer to that variable.

Local Variable Names: Can be referenced from point of definition to end of {…}

```
static void silly(int m) {              m
    int i = 4;                          m     i
                                        m     i
    for(int j=0; j<10; j++) {           m     i     j
       int k = 2;                       m     i     j     k
       k = k + i + j;                   m     i     j     k
    }                                   m     i
                                        m     i
    for(int j=0; j<20; j++) {           m     i     j
       int k = 9;                       m     i     j     k
       k = k + i - j;                   m     i     j     k
    }                                   m     i
                                        m     i
}
```

Digression: Scope in Java

The scope of a variable (e.g., local variable, parameter) is the region of the program where it is legal to refer to that variable.

Member names: Can be referenced ANYWHERE in the class and from outside if public

```
public class TestDefault {
       int n;                          n      m      k      p
       int m = 4;                      n      m      k      p
                                       n      m      k      p
       int sillyMethod(int q) {        n      m      k      p      q
          return q + n + m + k;        n      m      k      p      q
       }                               n      m      k      p
                                       n      m      k      p
       int k = n + m;                  n      m      k      p
       int p = m + 1;                  n      m      k      p
  }
```

Scope in Labda Expressions

The **scope of a lambda parameter** is the expression to the right of the **->**

```
(\x -> <expression>)
```

Scope of `x`

To find the parameter associated with an instance of a variable in the expression, look for the **closest enclosing binding of the variable**:

```
(\x -> \ys -> (length (take x ys)))
```

Scope in Lambda Expressions: Hole in Scope

To find the parameter associated with an instance of a variable in the expression, look for the **closest enclosing binding of the variable:**

```
(\x -> \ys -> (length (take x ys)))
```

Some weird things can happen when there is more than one occurrence of the same variable:

```
Main> (\x -> ((\x ->  (take x [1,2,3,4,5]) ) 3) ++ x) [7]
[1,2,3,7]
```

```
Main>(\x -> (\x -> (x,x))) 5 True
(True,True)
```

Hole in scope of outer x

Digression: Scope in Java

Java allows multiple declarations of the same variable if one is a field and one is a local variable (either a parameter or a local variable), creating a hole in the scope of the field declaration:

```java
public class Test
{
    public int x = 1;

    public static void f(int x) {
        // int x = 2
        System.out.println(x);
       // for(int x = 10; x <15; ++x) {
       //      System.out.println(x);
       // }
    }

    public static void main(String args[])
    {
        {
        //  int x = 3;
            {
                int x = 4;
                System.out.println(x);

                f(5);
            }
        }
    }
}
```

```
$javac Test.java
$java -Xmx128M -Xms16M Test
4
5
```

Digression: Scope in Java

But Java does NOT allow multiple declarations (and hence avoids the hole in scope issue) for two local variables:

```java
public class Test
{
    public int x = 1;

    public static void f(int x) {
        int x = 2;
        System.out.println(x);
      // for(int x = 10; x <15; ++x) {
      //     System.out.println(x);
      // }
    }

    public static void main(String args[])
    {
        {
        //  int x = 3;
            {
                int x = 4;
                System.out.println(x);

                f(5);
            }
        }
    }
}
```

```
$javac Test.java
Test.java:6: error: variable x is already defined in method f(int)
        int x = 2;
            ^
1 error
```

Digression: Scope in Java

But Java does NOT allow multiple declarations (and hence avoids the hole in scope issue) for two local variables:

```java
1  public class Test
2  {
3      public int x = 1;
4
5      public static void f(int x) {
6          //int x = 2;
7          System.out.println(x);
8          for(int x = 10; x <15; ++x) {
9             System.out.println(x);
10         }
11     }
12
13     public static void main(String args[])
14     {
15         {
16         //  int x = 3;
17             {
18                 int x = 4;
19                 System.out.println(x);
20
21                 f(5);
22             }
23         }
24     }
25 }
26
```

```
$javac Test.java
Test.java:8: error: variable x is already defined in method f(int)
        for(int x = 10; x <15; ++x) {
                ^
1 error
```

Digression: Scope in C

C allows multiple declarations without many restrictions:

```c
#include <stdio.h>

int x = 5;

int main()
{
    int x = 1;

    if (x == 1)
        printf("x is equal to one.\n");
    else
        printf("x is not equal to one.\n");

    return 0;
}
```

```
is equal to one.
```

```c
#include <stdio.h>

int x = 5;

int main()
{
    int x = 1;
    {
        int x = 3;
        if (x == 1)
            printf("x is equal to one.\n");
        else
            printf("x is not equal to one.\n");
    }

    return 0;
}
```

```
x is not equal to one.
```

Let Expressions in Haskell

In Haskell we create local variables using let:

```
(let x = <expr1> in <expr2>)
```

```
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r ^2
    in  sideArea + 2 * topArea
```

Scope of local variables

Equivalent to a lambda application:

```
((\x -> <expr2>) <expr1>)
```

Except that you can have multiple bindings in the same let.

```
    let sq x = x * x in (sq 5, sq 3, sq 2)

 => (25,9,4)

    let x = 5
    in let y = 2 * x
       in let z = x + y
          in (\w -> x * y + z) 10

 => 65
```

Let Expressions in Haskell

Haskell let's you define local variables any time you want with let (and where), and therefore hole in scope issues become relevant.

Notice the great flexibility of Haskell and the referential transparency principle: You can use these kinds of expressions nearly anywhere!

```
    (let sq = (\x -> x*x) in \x -> (x,sq x) ) 5

 => (5,25)

     (\x -> case x of
                 1 -> \x -> x + 1
                 2 -> \x -> x * 2
                 _ -> \x -> x      ) 2 6

  =>  12
```

Modules

"A **Haskell module is a collection of related functions, types and typeclasses**. A Haskell program is a collection of modules where the main module loads up the other modules and then uses the functions defined in them to do something. Having code split up into several modules has quite a lot of advantages. If a module is generic enough, the functions it exports can be used in a multitude of different programs. If your own code is separated into self-contained modules which don't rely on each other too much (we also say they are loosely coupled), you can reuse them later on. It makes the whole deal of writing code more manageable by having it split into several parts, each of which has some sort of purpose." – Learn You a Haskell .....

Using modules

```
import Prelude                          -- Import everything from the module Prelude
                                        -- If you have no imports, Prelude is imported
                                        -- by default.

import Prelude  (Show,undefined)        -- Import ONLY Show and undefined

import Prelude hiding (map, filter)     -- Import everything EXCEPT map and filter
```

Modules

Creating modules

For now, just remember to put all modules in the same directory as the code where they will be imported.....

Use the following syntax in the first line of your file to create a module; the name must be the same as the file (without the .hs):

Test.hs

```haskell
module Test where

-- this module allows anything defined in the module to be
-- visible outside the module.
```

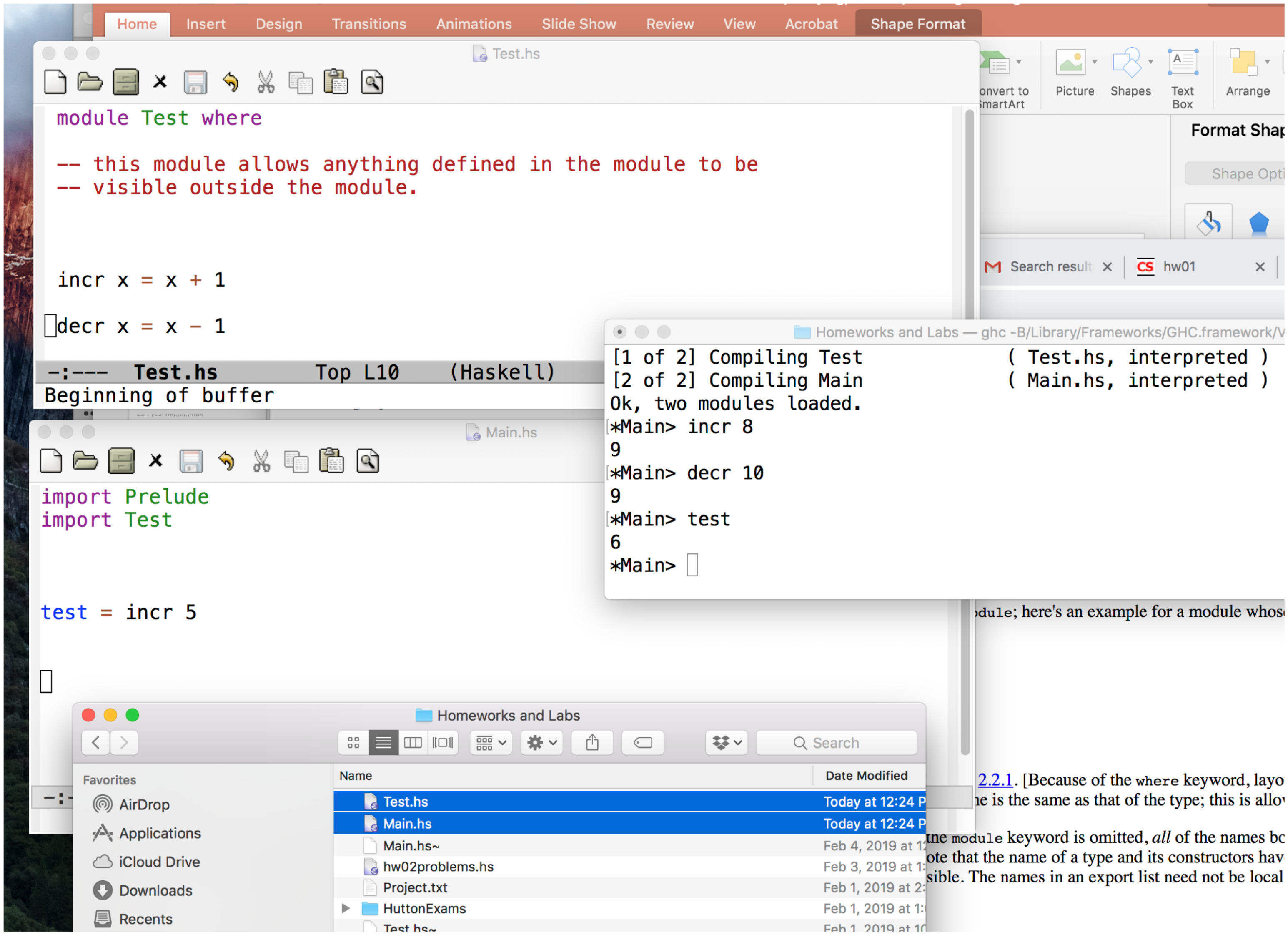
Test.hs

```haskell
module Test (map, filter)  where

-- this module only allows map and filter to be visible outside the module
```

There is no way to hide only some names from being exported from a module. You have to list the names you DO want to export. You can only using the keyword **hiding** in an import statement.

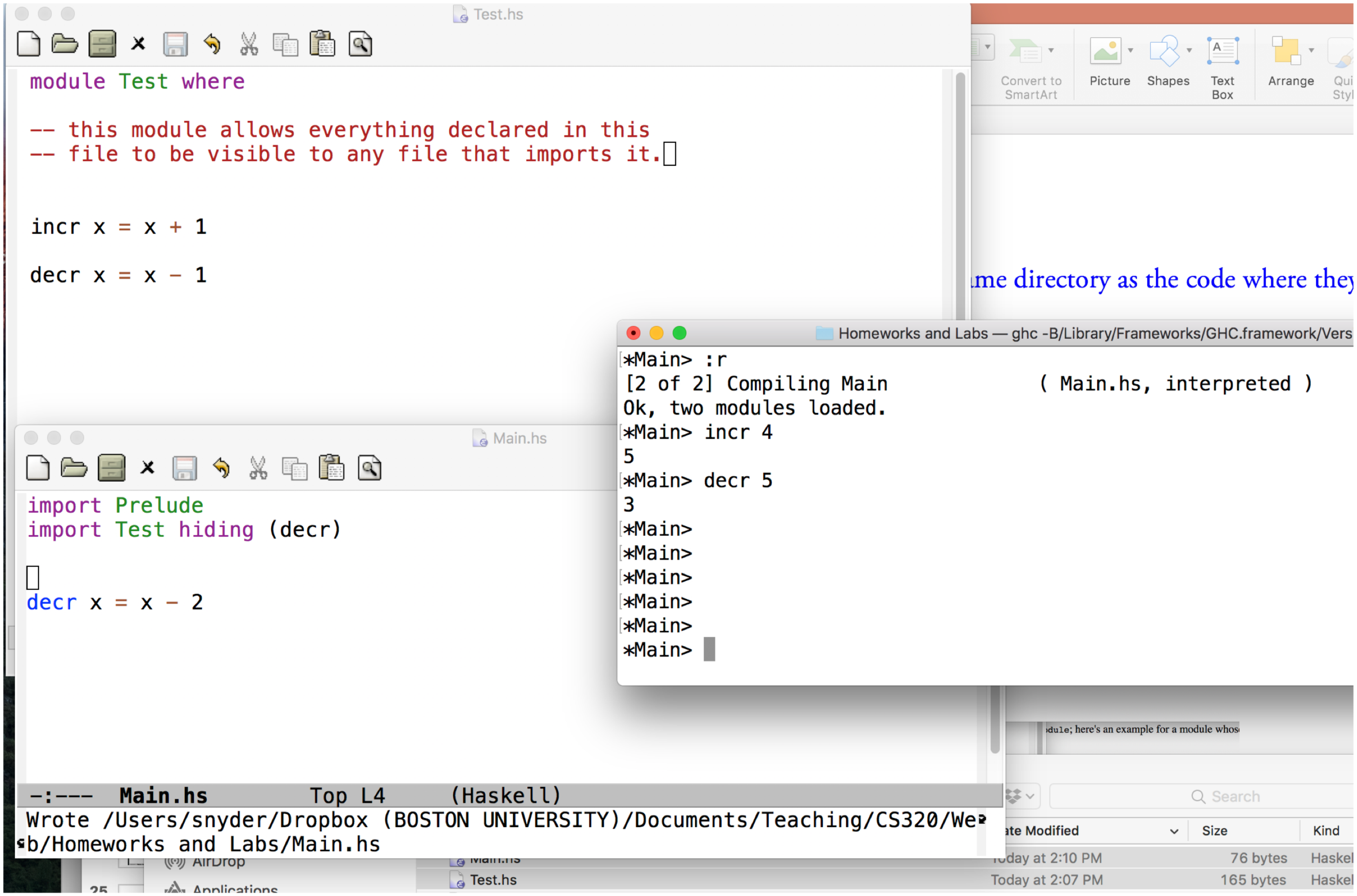
Modules

For now, just remember to put all modules in the same directory as the code where they will be imported.....



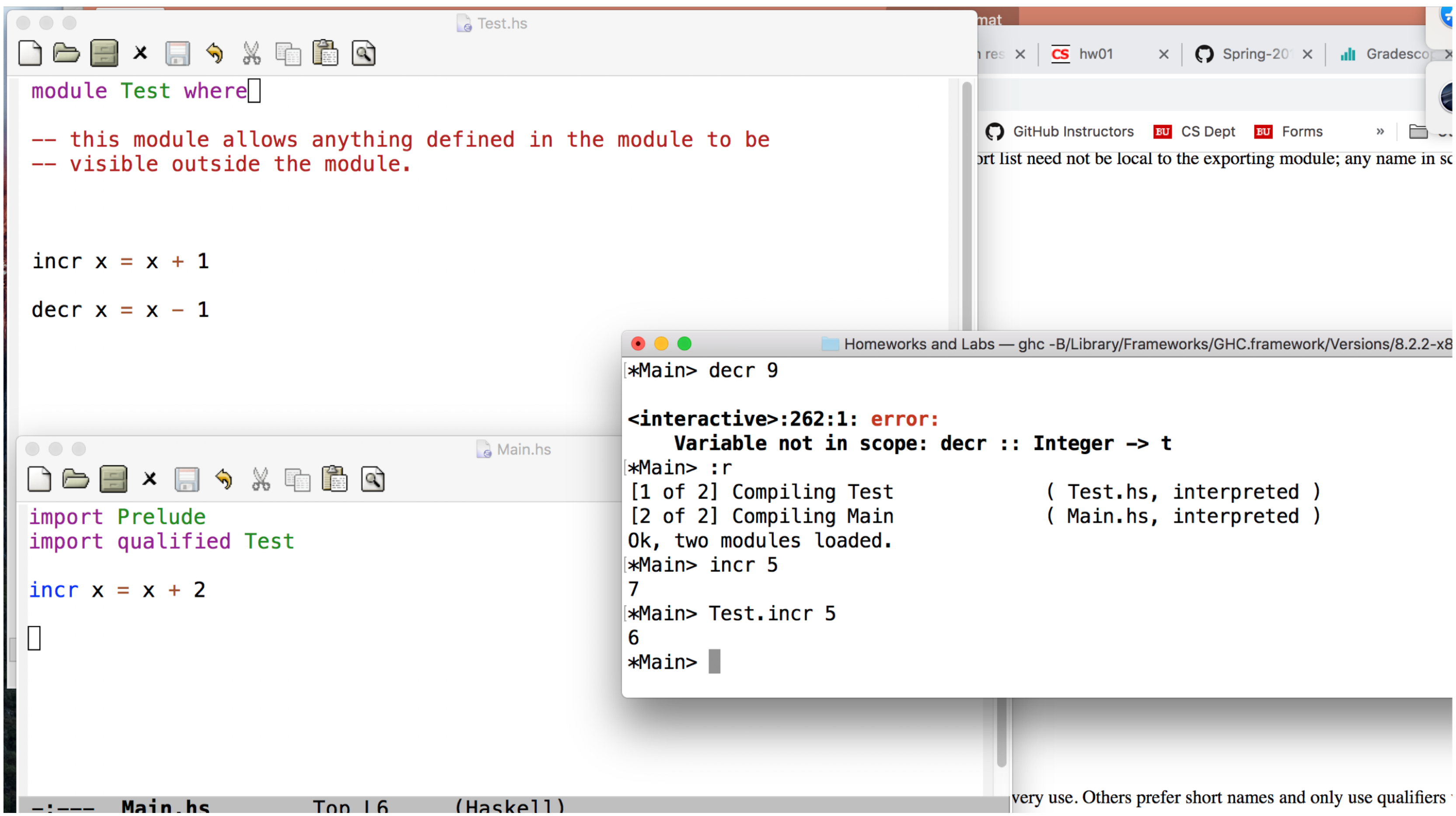
Modules

For now, just remember to put all modules in the same directory as the code where they will be imported.....



Modules: Qualified Imports

"There is an obvious problem with importing names directly into the namespace of module. What if two imported modules contain different entities with the same name? Haskell solves this problem using *qualified names*. An import declaration may use the qualified keyword to cause the imported names to be prefixed by the name of the module imported. These prefixes are followed by the `.' character without intervening whitespace."
– https://www.haskell.org/tutorial/modules.html



Modules: Qualified Imports with Local Names