

CS 320: Concepts of Programming Languages

Wayne Snyder
Computer Science Department
Boston University

Lecture 05: 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 extent 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:

```
data List a = Nil | Cons a (List a) deriving Show

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

incrList :: List Integer -> List Integer
incrList Nil = Nil
incrList (Cons x xs) = (Cons (incr x) (incrList xs))
```

```
Main> incrList (Cons 3 (Cons 5 Nil))
Cons 4 (Cons 6 Nil)
```

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 = mod x 2 == 0
```

```
isEvenList :: List Integer -> List Bool
isEvenList Nil = Nil
isEvenList (Cons x xs) = (Cons (isEven x) (isEvenList xs))
```

```
Main> isEvenList (Cons 3 (Cons 5 Nil))
Cons False (Cons False Nil)
```

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 :: List Integer -> List Integer
incrList Nil = Nil
incrList (Cons x xs) = (Cons (incr x) (incrList xs))

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

isEvenList :: List Integer -> List Bool
isEvenList Nil = Nil
isEvenList (Cons x xs) = (Cons (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 = mod x 2 == 0
```

```
isEvenList :: List Integer -> List Bool
isEvenList Nil = Nil
isEvenList (Cons x xs) = (Cons (isEven x) (isEvenList xs))
```

```
map :: (a -> b) -> List a -> List b
map f Nil = Nil
map f (Cons x xs) = (Cons (f x) (map f xs))
```

Functional Programming Paradigms

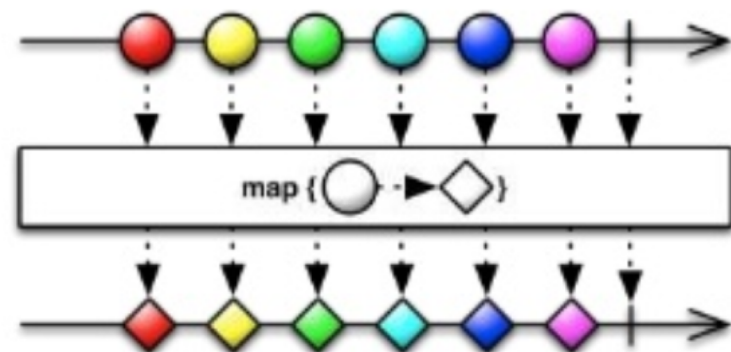
- passed as parameters
- stored in data structures
- represented as values without having to assign to a name.
- manipulated by other functions

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

```
map :: (a -> b) -> List a -> List b
map f Nil = Nil
map f (Cons x xs) = (Cons (f x) (map f xs))
```

```
Main> map incr (Cons 3 (Cons 5 Nil))
Cons 4 (Cons 6 Nil)
```

```
Main> map times2 (Cons 3 (Cons 5 Nil))
Cons 6 (Cons 10 Nil)
```



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 = mod x 2 == 0
```

```
filterEvenList :: List Integer -> List Integer
filterEvenList Nil = Nil
filterEvenList (Cons x xs) | isEven x = (Cons x (filterEvenList xs))
                           | otherwise = (filterEvenList xs)
```

```
Main> filterEvenList (Cons 2 (Cons 3 (Cons 4 Nil)))
Cons 2 (Cons 4 Nil)
```


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 = mod x 2 == 0
```

```
filterEvenList :: List Integer -> List Integer
filterEvenList Nil = Nil
filterEvenList (Cons x xs) | isEven x = (Cons x (filterEvenList xs))
                           | otherwise = (filterEvenList xs)
```

```
filter :: (a -> Bool) -> List a -> List a
filter p Nil = Nil
filter p (Cons x xs) | p x = (Cons x (filter p xs))
                    | otherwise = (filter p xs)
```

```
Main> filter isEven (Cons 2 (Cons 3 (Cons 4 Nil)))
Cons 2 (Cons 4 Nil)
```


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 :: List (a -> b) -> List a -> List b
applyList Nil _ = Nil
applyList _ Nil = Nil
applyList (Cons f fs) (Cons x xs) = Cons (f x) (applyList fs xs)
```

```
Main> funcList = (Cons incr (Cons times2 (Cons decr Nil)))
```

```
Main> argList = (Cons 4 (Cons 5 (Cons 9 Nil)))
```

```
Main> applyList funcList argList
Cons 5 (Cons 10 (Cons 8 Nil))
```

```
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 = (Cons incr (Cons times2 (Cons decr Nil)))
```

```
Main> argList = (Cons 4 (Cons 5 (Cons 9 Nil)))
```

```
Main> head (Cons x _ ) = x
```

```
Main> tail (Cons _ xs) = xs
```

```
Main> f = head funcList
```

```
Main> f 8
```

```
9
```

```
Main> applyList (tail funcList) (tail argList)
```

```
Cons 10 (Cons 8 Nil)
```

```
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      (Cons 5 Nil)   'a'      "Hi there"
```

```
Main> x = 3
```

```
Main> lst = (Cons 5 Nil)
```

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 expressions** 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:

Or math notation:

$\lambda x. x+1$

```
script.py  IPython Shell
1  # Program to show the use of lambda functions
2
3  double = lambda x: x * 2
4
5  # Output: 10
6  print(double(5))
```

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> (\(Pair x y) -> x + y) (Pair 3 4)
```

```
7
```

```
Main> (\(Cons x xs) -> 2*x) (Cons 2 (Cons 3 (Cons 4)))
```

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

```
4
```

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) (Cons 2 (Cons 3 (Cons 4 Nil)))  
(Cons 3 (Cons 4 (Cons 5 Nil)))
```

```
Main> filter (\x -> mod x 2 == 0) (Cons 2 (Cons 3 Nil))  
(Cons 2 Nil)
```

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

```
Main> funcList = (Cons (\x -> x + 1) (Cons (\z -> z * 2) Nil))
```

```
Main> applyList funcList (Cons 2 (Cons 5 Nil))  
(Cons 3 (Cons 10 Nil))
```


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:

$$f \circ g (x) = f (g(x))$$

Function composition in Haskell:

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

```
plus1times2 = times2 . incr
```

```
Main> incr 2
```

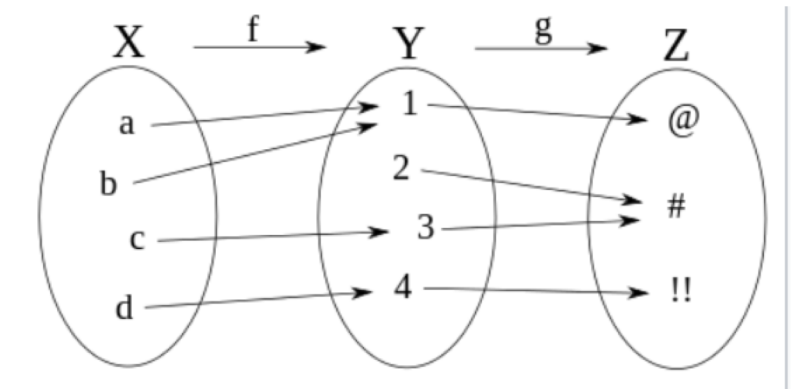
```
3
```

```
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 (Cons 2 (Cons 4 Nil))
```

```
(Cons 2 (Cons 4 (Cons 0 Nil)))
```

```
Main> map (`div` 2) (Cons 4 (Cons 7 Nil))
```

```
Cons 2 (Cons 3 Nil)
```

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

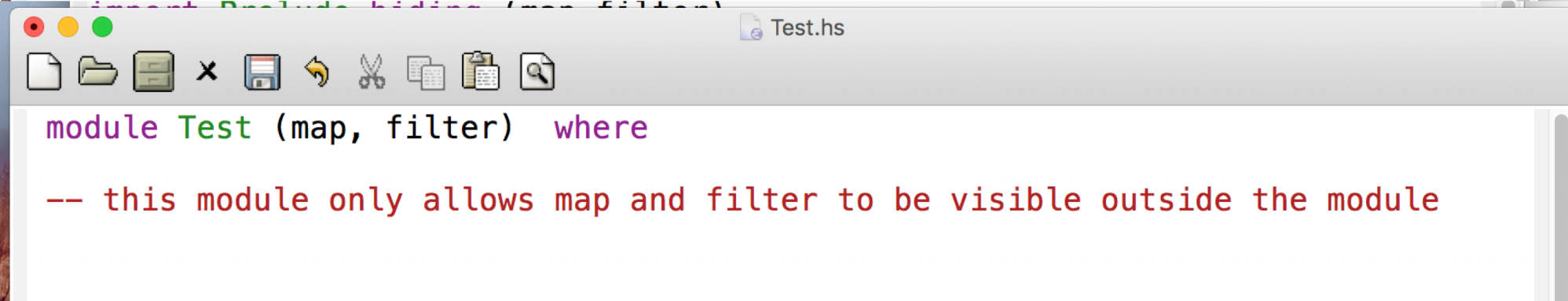
For now, just remember to put all modules in the same directory as the code where they will be imported.....

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):



```
module Test where
```

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



```
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.....

The screenshot shows a Haskell development environment. The top window is a code editor for `Test.hs` with the following content:

```
module Test where

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

incr x = x + 1
decr x = x - 1
```

The bottom window is a code editor for `Main.hs` with the following content:

```
import Prelude
import Test

test = incr 5
```

The middle window is a terminal window showing the compilation and execution of the code:

```
[1 of 2] Compiling Test          ( Test.hs, interpreted )
[2 of 2] Compiling Main          ( Main.hs, interpreted )
Ok, two modules loaded.
*Main> incr 8
9
*Main> decr 10
9
*Main> test
6
*Main>
```

The bottom window is a file browser showing the directory `Homeworks and Labs` with the following files:

Name	Date Modified
Test.hs	Today at 12:24 P
Main.hs	Today at 12:24 P
Main.hs~	Feb 4, 2019 at 1
hw02problems.hs	Feb 3, 2019 at 1
Project.txt	Feb 1, 2019 at 2
HuttonExams	Feb 1, 2019 at 1
Test.hs~	Feb 1, 2019 at 10

Partial text from the right side of the image:

...dule; here's an example for a module whose

2.2.1. [Because of the `where` keyword, layo
... is the same as that of the type; this is allo

...the `module` keyword is omitted, *all* of the names be
...ote that the name of a type and its constructors hav
...sible. The names in an export list need not be local

Modules

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

The screenshot shows a Haskell IDE with two code editors and a terminal window. The top editor, titled 'Test.hs', contains the following code:

```
module Test where

-- this module allows everything declared in this
-- file to be visible to any file that imports it.

incr x = x + 1
decr x = x - 1
```

The bottom editor, titled 'Main.hs', contains the following code:

```
import Prelude
import Test (incr)

decr x = x - 2
```

The terminal window, titled 'Homeworks and Labs — ghc -B/Library/Frameworks/GHC.framework/V...', shows the following output:

```
6
*Main> incr2 5
7
*Main>
*Main> :r
[2 of 2] Compiling Main
Ok, two modules loaded.
*Main>
*Main> incr 4
5
*Main> decr 5
3
*Main>
```

The status bar at the bottom of the IDE shows: `--:--- Main.hs Top L5 (Haskell)` and `(No changes need to be saved)`. The system tray at the bottom right shows the date and time: `today at 2:09 PM` and `69 bytes`.

Modules

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

The image shows a Haskell code editor with two files: `Test.hs` and `Main.hs`. The `Test.hs` file defines a module `Test` with functions `incr` and `decr`. The `Main.hs` file imports `Prelude` and `Test` (hiding `decr`), and defines a `decr` function. A terminal window shows the execution of these functions, resulting in the output `5` and `3`.

```
module Test where

-- this module allows everything declared in this
-- file to be visible to any file that imports it.

incr x = x + 1

decr x = x - 1
```

```
import Prelude
import Test hiding (decr)

decr x = x - 2
```

```
*Main> :r
[2 of 2] Compiling Main
Ok, two modules loaded.
(Main.hs, interpreted )
*Main> incr 4
5
*Main> decr 5
3
*Main>
*Main>
*Main>
*Main>
*Main>
*Main>
```

File list:

Date Modified	Size	Kind
Today at 2:10 PM	76 bytes	Haskell
Today at 2:07 PM	165 bytes	Haskell

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>

The image shows a screenshot of a Haskell development environment. It features two code editors and a terminal window. The top editor, titled 'Test.hs', contains the following code:

```
module Test where

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

incr x = x + 1
decr x = x - 1
```

The bottom editor, titled 'Main.hs', contains the following code:

```
import Prelude
import qualified Test

incr x = x + 2

[]
```

The terminal window, titled 'Homeworks and Labs — ghc -B/Library/Frameworks/GHC.framework/Versions/8.2.2-x8', shows the following output:

```
*Main> decr 9
<interactive>:262:1: error:
  Variable not in scope: decr :: Integer -> t
*Main> :r
[1 of 2] Compiling Test           ( Test.hs, interpreted )
[2 of 2] Compiling Main           ( Main.hs, interpreted )
Ok, two modules loaded.
*Main> incr 5
7
*Main> Test.incr 5
6
*Main>
```

At the bottom of the image, there is a footer that reads: "very use. Others prefer short names and only use qualifiers".

Modules: Qualified Imports with Local Names

The image shows a Haskell development environment with three windows:

- Test.hs:** A module definition for `Test`.

```
module Test where

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

incr x = x + 1
decr x = x - 1
```
- Main.hs:** A module that imports `Test` with a qualified name.

```
import Prelude
import qualified Test as T

incr x = x + 2
```
- Terminal:** A GHCi session showing the execution of the code.

```
*Main> incr 5
7
*Main> Test.incr 5
6
*Main> :r
[2 of 2] Compiling Main           ( Main.hs, interpreted)
Ok, two modules loaded.
*Main>
*Main>
*Main>
*Main> T.incr 5
6
*Main>
```

At the bottom, a status bar shows the current file is `Main.hs` at line 2, column 20, in the Haskell mode. Below it, a snippet of text explains the behavior of qualified imports: "allowed: an entity can be imported by various routes without conflict. The compiler knows whether entities from different modules are actually the same".