Functions as Values

week	date	Monday	Tuesday	Thursday
1	Jan. 9	**Introduction**	Haskell Start-Up	Haskell Start-Up
2	Jan 16	Haskell Start-Up	Recursion	Lists and Tuples *(assn 1 due)*
3	Jan 23	More About Lists	Proofs	I/O
4	Jan 30	Algebraic Types	**Quiz 1**	Algebraic Types *(assn 2 due)*
5	Feb 6	Generalization	**Functions As Values**	Type Classes & Checking
6	Feb 13	Lazy Programming	Haskell overflow	Haskell overflow
7	Feb 27	Haskell review	**Quiz 2**	Prolog...
8	Mar 6	*(assn 3 due)*		
9	Mar 13		*(assn 4 due)*	
10	Mar 20		**Quiz 3**	
11	Mar 27		*(assn 5 due)*	
12	Apr 3		Prolog overflow	Prolog overflow *(assn 6 due)*

What's This Topic About?

Different ways of creating functions
or
Expressions whose values are functions

Frequent motivation: avoid having to define trivial helper functions

Techniques:

- function composition
- partial application of functions & operators
- lambda notation
- currying & uncurrying

Required Reading: Chapter 10 (skip section 10.9)

Function Composition

Operator "." composes two functions
means apply in sequence

Example: find second element in list

```
second :: [a] -> a
second = head . tail
```


Operator "Sections"

```
incrList :: [Int] -> [Int]
incrList nums = map add1 nums
  where
  add1 x = x+1
```

Used a very simple helper function. There's a quicker way.

Recall: **(+)** is the function that adds two numbers together.

Operator Section: **(1+)** is the function that adds 1 to a number

Operator Sections (2)

```
-- new, simpler definition
incrList nums = map (1+) nums
```

To create unary function from binary operator – can supply either first or second operator

Makes no difference with "+", does with other operators

Example:

- `(/10)`: function that divides its parameter by 10
- `(10/)`: function that divides 10 by its parameter

```
map (10 /) [2.0,5.0] =  [5.0,2.0]

map (/ 10) [2.0,5.0] =  [0.2,0.5]
```


More Examples

```
filter (/=0) [1,0,-2,5,0]  [1,-2,5]

map (++"!!!") ["hello","world"]
      ["hello!!!","world!!!"]

map ((*3).(+2)) [1,2,3]    [9,12,15]

map (`mod` 10) [43, 57, 92] [3,7,2]
```


Partial Function Application

Consider this definition:

```
f :: Int -> Int -> Int
f x y = (3*x) + (2*y)
```

What's the meaning of `(f 2)`?

Equivalent to **g**, where:

```
g y = 6 + (2*y)

map (f 2) [1,2,3] =  [8,10,12]
```


Example

Recall Prelude function zip:

```
  zip :: [a] -> [b] -> [(a,b)]
```

```
concat
  (map (zip [1..20])
    ["Mary","had","a","little","lamb"])
```

```
[(1,'M'),(2,'a'),(3,'r'),(4,'y'),(1,'h'),
 (2,'a'),(3,'d'),(1,'a'),(1,'l'),(2,'i'),
 (3,'t'),(4,'t'),(5,'l'),(6,'e'),(1,'l'),
 (2,'a'),(3,'m'),(4,'b')]
```

Another Example

Recall Prelude function **drop**:

```
   drop :: Int -> [a] -> [a]
```

Example: `drop 2 "abcd" = "cd"`

```
map (drop 3) ["Mickey","Mouse","Club"]

["key","se","b"]
```


Order Of Parameters

What if you wanted to fix the *second* parameter of drop?
Example: successive tails of a list

```
tails :: [a] -> [[a]]
tails lis = map helper [0..(length lis)]
  where
  helper n = drop n lis
tails "abc" =  ["abc","bc","c",""]
```

Can't use partial function application directly to replace helper.
Two options:

```
   map (`drop` lis) [0..(length lis)]

   map ((flip drop) lis) [0..(length lis)]
```


Lambda Notation

Sometimes operator sections & partial function application isn't enough to eliminate trivial helper function
Example:

```
squareList :: [Int] -> [Int]
squareList lis = map square lis
  where
  square n = n*n
```

Lambda notation lets us define small anonymous functions

```
squareList lis = map (\n->n*n) lis
```

`(\n->n*n)` means:
"A function that takes one parameter and multiplies it by itself"

Lambda Notation With Multiple Parameters

```
\x y -> sqrt (x*x + y*y)
```

means:
A function that takes 2 sides of a right triangle and returns the hypotenuse

Two equivalent ways to give this a name:

```
hypot1 = \x y -> sqrt (x*x + y*y)
hypot2 x y = sqrt (x*x + y*y)
```

Another Example

Problem: Given three numbers a, b and c, create a function to evaluate the quadratic ax^2+bx+c

Solution:

```
quad a b c = \x-> a*x*x + b*x + c
```

Equivalent Solution:

```
quad a b c x = a*x*x + b*x + c
```

Using quad to evaluate x^2+2x+3 for x = 2

(quad 1 2 3) 2

or:

quad 1 2 3 2

These two expressions mean the same thing!
Function application associates to the left

Digression: Who Was Haskell?

Haskell Brooks Curry
(1900-1982)
Well-known mathematical logician

Haskell language named after him.
Also concept of "curried functions"

Curried Functions

Most functions we've looked at this term have been "curried".
Simple example of a curried function:

```
f :: Int -> Int -> Int
f x y = (3*x) + (2*y)
```

- **f** takes its parameters one at a time.
- **f x** produces a function with one parameter.
- **(f x) y** produces a numerical value.
- Parenthesis not necessary: can write **f x y**
- We usually think of f as having two parameters.

Uncurried Functions

An uncurried function combines all its parameters into a tuple.
(Technically, just one parameter)

Uncurried:

```
f (x,y) = (3*x) + (2*y)
```

Curried:

```
f x y = (3*x) + (2*y)
```

curry & uncurry functions

higher-order functions to move between curried & uncurried functions of two parameters

```
uncurry :: (a->b->c) -> ((a,b)->c)
```

Given:

```
f x y = (3*x) + (2*y)
g = uncurry f

g (1,2) = 3*1 + 2*2 = 7
```

curry does the opposite:

```
curry :: ((a,b)->c) -> (a->b->c)
(curry g) is equivalent to f
```


Example

Problem: a list of tuples, want to add them all together

Example: `[(1,3),(2,7),(3,2)] -> [4,9,5]`

```
sumList lis = map (uncurry (+)) lis
```