Creating Functions

Functional Programming

The function calculator

- Functional programming is all about using functions
- Functions are first class
  - Take as input, return as result, store in data
- A functional language is a function calculator
- What buttons do we have for “creating” functions?

12 ways to get a new function

- By defining one at top level
  - By equation
  - By cases
  - By patterns
- By local definition (where and let)
- By use of a library
- By lambda expression (anonymous functions)
- By parenthesizing binary operators
- By section
- By currying (partial application)
- By composition
- By combinator (higher order functions)
- By using data and lookup (arrays lists and finite functions)

By defining at top level

```
Module Test where
plus5 x = x + 5

last x = head(reverse x)
```

```
CreatingFunctions> plus5 7
12
CreatingFunctions> last [2,3,4]
4
```

By cases

Vertical Bar

Equal sign

```
absolute x  | x < 0   =  -x
            | x >= 0  =  x
```

Name and argument

condition

Value for case

```
swap (x,y)  | x < y  =  (x,y)
            | x > y  =  (y,x)
            | x==y   =  (x,y)
```

```
CreatingFunctions> absolute 3
3
CreatingFunctions> absolute (-4)
4
CreatingFunctions> swap (23,5)
(5,23)
```

By patterns

- Example on Booleans

```
myand True False = False
myand True True = True
myand False False = False
myand False True = False
```

Pattern may contain constructors. Constructors are always capitalized. True and False are constructors

- Order Matters
  - Variables in Patterns match anything

```
myand2 True True = True
myand2 x y = False
```

  - What happens if we reverse the order of the two equations above?

By local definition

(where and let)

```
ordered = sortBy backwards
                    [1,76,2,5,9,45]
  where backwards x y = compare y x
```

```
CreatingFunctions> ordered
[76,45,9,5,2,1]
```

By use of a Library

```
smallest = List.minimum
                       [3,7,34,1]
```

```
CreatingFunctions> smallest
1
```

By lambda expression

(anonymous functions)

```
CreatingFunctions> descending
[76,45,9,5,2,1]
CreatingFunctions> bySnd
[[(1,'a'),(3,'a')],[(2,'c')]]
```

```
descending =
   sortBy
   (\ x y -> compare y x)
   [1,76,2,5,9,45]
bySnd =
  groupBy
  (\ (x,y) (m,n) -> y==n)
  [(1,'a'),(3,'a'),(2,'c')]
```

By parenthesizing binary operators

```
six:: Integer
-- 1 + 2 + 3 + 0
six = foldr (+) 0 [1,2,3]
```

```
CreatingFunctions> six
6
```

By section

```
add5ToAll = map (+5) [2,3,6,1]
```

```
CreatingFunctions> add5ToAll
[7,8,11,6]
```

By partial application

Note, both map and any, each take 2 arguments

```
hasFour = any (==4)
doubleEach = map (\ x -> x+x)
```

```
CreatingFunctions> hasFour
[2,3]
False
CreatingFunctions> hasFour
[2,3,4,5]
True
CreatingFunctions> doubleEach
[2,3,4]
[4,6,8]
```

By composition

```
hasTwo = hasFour . doubleEach
empty = (==0) . length
```

```
CreatingFunctions> hasTwo
[1,3]
False
CreatingFunctions> hasTwo
[1,3,2]
True
CreatingFunctions> empty [2,3]
False
CreatingFunctions> empty []
True
```

By combinator

(higher order functions)

```
k x = \ y -> x

all3s = map (k 3) [1,2,3]
```

```
CreatingFunctions> :t k True
k True :: a -> Bool
CreatingFunctions> all3s
[3,3,3]
```

Using data and lookup

(arrays, lists, and finite functions)

```
whatDay x =
  ["Sun","Mon","Tue","Wed","Thu","Fri","Sat"]
  !! x


first9Primes =  array (1,9)
    (zip [1..9]
         [2,3,5,7,11,13,17,19,23])


nthPrime x = first9Primes ! x
```

```
CreatingFunctions> whatDay 3
"Wed"
CreatingFunctions> nthPrime 5
11
```

When to define a higher order function?

- Abstraction is the key

```
mysum [] = 0
mysum (x:xs) = (+) x (mysum xs)
myprod [] = 1
myprod (x:xs) = (*) x (myprod xs)
myand [] = True
myand (x:xs) = (&&) x (myand xs)
```

- Note the similarities in definition and in use

```
? mysum [1,2,3]
6
? myprod [2,3,4]
24
? myand [True, False]
False
```

When do you define a higher order function?

- Abstraction is the key

```
mysum [] = 0
mysum (x:xs) = (+) x (mysum xs)
myprod [] = 1
myprod (x:xs) = (*) x (myprod xs)
myand [] = True
myand (x:xs) = (&&) x (myand xs)
```

- Note the similarities in definition and in use

```
? mysum [1,2,3]
6
? myprod [2,3,4]
24
? myand [True, False]
False
```

Abstracting

```
myfoldr op e [] = e
myfoldr op e (x:xs) =
      op x (myfoldr op e xs)
```

```
? :t myfoldr
myfoldr :: (a -> b -> b) -> b -> [a] -
  > b
? myfoldr (+) 0 [1,2,3]
6
?
```

Functions returned as values

- Consider:

```
k x = (\ y -> x)

? (k 3) 5
3
```

- Another Example:

```
plusn n = (\ x -> x + n)

? (plusn 4) 5
9
```

- Is `plusn` different from `plus`? why?
  - `plus x y = x + y`