

Objectives

Introduction to Higher Order Functions

Dr. Mattox Beckman

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
DEPARTMENT OF COMPUTER SCIENCE

- ▶ Explain the concept of *first class citizen*.
- ▶ Use sectioning and lambda to define anonymous functions.
- ▶ Change the behavior and interface of a function by using another function.

First Class Functions

An entity is said to be *first class* when it can be:

- ▶ **Assigned** to a variable, **passed** as a parameter, or **returned** as a result

Examples:

- ▶ APL: scalars, vectors, arrays
- ▶ C: scalars, pointers, structures
- ▶ C++: like C, but with objects
- ▶ HASKELL, LISP, OCAML: scalars, lists, tuples, functions

The Kind of Data a Program Manipulates Changes the Expressive Ability of a Program.

Defining Functions the Usual Way

Some HASKELL Functions

```
1 sqr a = a * a
2 hypotsq a b = sqr a + sqr b
```

Sample Run

```
1 sqr :: Integer -> Integer
2 sqr :: Num a => a -> a
3 hypotsq :: Num a => a -> a -> a
4 Prelude> sqr 10
5 100
6 Prelude> hypotsq 3 4
7 25
```

Example: Compose

Example

```
1 inc x = x + 1
2 double x = x * 2
3 compose f g x = f (g x)
```

- ▶ Notice the function types.

```
1 compose ::  $\overbrace{(t1 \rightarrow t2)}^f \rightarrow \overbrace{(t \rightarrow t1)}^g \rightarrow t \rightarrow t2$ 
2 Prelude> :t double
3 double :: Integer -> Integer
4 Prelude> double 10
5 20
6 Prelude> compose inc double 10
7 21
```

◀ ▶ ⏪ ⏩ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹ ⏺ ⏻ ⏼ ⏽ ⏾ ⏿ 🔍 ↺

Creating Functions: Lambda Form

- ▶ Functions do not have to have names.

```
1 \x -> x + 1
```

- ▶ The parts:

- ▶ Backslash (a.k.a. *lambda*)
- ▶ Parameter list
- ▶ Arrow
- ▶ Body of function

```
1 prelude> (\x -> x + 1) 41
2 42
```

◀ ▶ ⏪ ⏩ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹ ⏺ ⏻ ⏼ ⏽ ⏾ ⏿ 🔍 ↺

Example: Twice

- ▶ One handy function allows us to do something twice.
- ▶ You will see this function again!

Twice

```
1 twice f x = f (f x)
```

Here is a sample run ...

```
Prelude> :t twice
twice :: (t -> t) -> t -> t
Prelude> twice inc 5
7
Prelude> twice twice inc 4
```

◀ ▶ ⏪ ⏩ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹ ⏺ ⏻ ⏼ ⏽ ⏾ ⏿ 🔍 ↺

Creating Functions: Partial Application

Standard Form vs. Anonymous Form

```
1 inc :: (Num t) => t -> t
2 inc a = a + 1
3 inc = \a -> a + 1
4
5 plus :: (Num t) => t -> t -> t
6 plus a b = a + b
7 plus = \a -> \b -> a + b
```

- ▶ What do you think we would get if we called `plus 1`?

◀ ▶ ⏪ ⏩ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹ ⏺ ⏻ ⏼ ⏽ ⏾ ⏿ 🔍 ↺

Creating Functions: Partial Application

Standard Form vs. Anonymous Form

```

1 inc :: (Num t) => t -> t
2 inc a = a + 1
3 inc = \a -> a + 1
4
5 plus :: (Num t) => t -> t -> t
6 plus a b = a + b
7 plus = \a -> \b -> a + b
    
```

► What do you think we would get if we called plus 1?

```

1 inc = plus 1
    
```



Curry and Uncurry

- Suppose you have a function tplus that takes a pair of integers and adds them.


```

1 tplus :: (Integer,Integer) -> Integer
2 tplus (a,b) = a + b
            
```
- But you really wish it took its arguments one at a time.
- There's a function curry :: (a,b) -> c -> a -> b -> c that will convert it for you! See if you can write it.



η-equivalence

An Equivalence

$$f \equiv \lambda x \rightarrow f x$$

► Proof, assuming f is a function...

$$f z \equiv (\lambda x \rightarrow f x) z$$

These are Equivalent

```

1 plus a b = (+) a b
2 plus a = (+) a
3 plus = (+)
    
```

So are These

```

1 inc x = x + 1
2 inc = (+) 1
3 inc = (+1)
    
```

