Basic Higher Order Functions

Dr. Mattox Beckman

Illinois Institute of Technology
Department of Computer Science

Objectives

You should be able to...

- Define *higher order function* and give some examples.
- Define the foldr and map functions.
- Use foldr and map to implement recursion patters we saw earlier.
- Understand the lambda form and how to use eta-expansion.

First Class Values

A type is said to be *first class* type 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 classes
- Scheme, Lisp, ML: scalars, lists, tuples, functions

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

Compose

Example

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

- Running the above code gives us...

```
Prelude> :t double
double :: (Num a) => a -> a
Prelude> :t compose
compose :: (t1 -> t2) -> (t -> t1) -> t -> t2
Prelude> compose inc double 10
21
```

Twice

- One handy function allows us to do something twice.

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
```


Lambda Form

- Functions do not have to have names.

These functions are equivalent

```
1 plus a b = a + b
2 plus' = \a -> \b -> a + b
```

- The two versions of `plus` are identical as far as the compiler is concerned.

η-expansion

An Equivalence

$$e \equiv \lambda x.e\ x$$

- Proof, assuming e is a function...

$$(\lambda x.e\ x)\ z \equiv e\ z$$

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)
```


Two Isomorphic Types

- Notice the difference between these two functions?

Door #1

```
foo a b = a + b
```

Door #2

```
bar (a,b) = a + b
```

Here is a sample run.

```
Prelude> foo 10 20
30
Prelude> bar (10,20)
30
Prelude> :t foo
foo :: (Num a) => a -> a -> a
Prelude> :t bar
bar :: (Num t) => (t, t) -> t
```

Curry

- A function that takes its arguments one at a time is called *curried*.[1]
- This function takes a non-curried function and returns a curried version of it!

curry

```
curry f a b = f (a,b)
```

```
Prelude> :t curry
curry :: ((t, t1) -> t2) -> t -> t1 -> t2
Prelude> :t curry bar
curry bar :: (Num t) => t -> t -> t
Prelude> curry bar 10 20
30
```

[1]Named after logician Haskell Curry.

Uncurry

- You can go the reverse definition.

uncurry

```
uncurry f (a,b) = f a b
```

Here is a possible use...

```
inc = curry plus 1
```


Other Examples

- Look at these and see if you understand what they do.

Example 1

```
ntimes 0 f x = x
ntimes n f x = f (ntimes (n-1) f x)
```

Example 2

```
flip f a b = f b a
```

Example 3

```
complist [] x = x
complist (f:fs) x = f $ complist fs x
```