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

<table>
<thead>
<tr>
<th>absolute x</th>
<th>x &lt; 0</th>
<th>-x</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x &gt;= 0</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>swap (x, y)</th>
<th>x &lt; y</th>
<th>(x, y)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x &gt; y</td>
<td>(y, x)</td>
</tr>
<tr>
<td></td>
<td>x==y</td>
<td>(x, y)</td>
</tr>
</tbody>
</table>

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

• Example on Booleans

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

• Order Matters
  – Variables in Patterns match anything

```plaintext
myand2 True True = True
myand2 x y = False
```
  – What happens if we reverse the order of the two equations above?

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

\[
\text{smallest} = \text{List.minimun} [3,7,34,1]
\]

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

\[ \text{six} :: \text{Integer} \]
\[ -- 1 + 2 + 3 + 0 \]
\[ \text{six} = \text{foldr} \ (+) \ 0 \ [1,2,3] \]
By section

add5ToAll = map (+5) [2,3,6,1]
By partial application

\[
\text{hasFour} = \text{any } (==4) \\
\text{doubleEach} = \text{map } (\times x \rightarrow x+x)
\]
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)

\[ k \ x = \ \lambda \ y \to x \]

\[ \text{all3s} = \text{map} \ (k \ 3) \ [1,2,3] \]

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
  
  \[
  \text{mysum} \; [] = 0 \\
  \text{mysum} \; (x:\text{xs}) = (+) \; x \; (\text{mysum} \; \text{xs})
  \]

  \[
  \text{myprod} \; [] = 1 \\
  \text{myprod} \; (x:\text{xs}) = (*) \; x \; (\text{myprod} \; \text{xs})
  \]

  \[
  \text{myand} \; [] = \text{True} \\
  \text{myand} \; (x:\text{xs}) = (\&\&) \; x \; (\text{myand} \; \text{xs})
  \]

• Note the similarities in definition and in use
  
  ? \text{mysum} \; [1,2,3] \\
  6

  ? \text{myprod} \; [2,3,4] \\
  24

  ? \text{myand} \; [\text{True}, \text{False}] \\
  \text{False}
When do you define a higher order function?

• Abstraction is the key

```haskell
mysum [] = 0
mysum (x:xs) = (+) x (mysum xs)
```

```haskell
myprod [] = 1
myprod (x:xs) = (*) x (myprod xs)
```

```haskell
myand [] = True
myand (x:xs) = (&&) x (myand xs)
```

• Note the similarities in definition and in use

```haskell
? mysum [1,2,3]
6

? myprod [2,3,4]
24

? myand [True, False]
False
```
Abstracting

myfoldr \( \text{op} \ e \ [\ ] = e \)
myfoldr \( \text{op} \ e \ (x:xs) = \)
\hspace{2cm} \text{op} \ x \ (\text{myfoldr} \ \text{op} \ e \ xs) \)

? :t myfoldr
myfoldr :: (a -> b -> b) -> b -> [a] -> b
? myfoldr (+) 0 [1,2,3] 6
?
Functions returned as values

• Consider:
  \( k \ x = (\ y \to \ x) \)

  \? (k 3) 5
  3
• Another Example:

  plusn \ n = (\ x \to x + n) \\

  \? (plusn 4) 5
  9
• Is \( \text{plusn} \) different from \( \text{plus} \)? why?
  \( \underline{\text{plus} \ x \ y = x + y} \)