

# Haskell — Curried Functions

Copyright © 2013 Christian Collberg

# Infix Functions

# Declaring Infix Functions

- Sometimes it is more natural to use an infix notation for a function application, rather than the normal prefix one:
  - $5 + 6$  (infix)
  - $(+) 5 6$  (prefix)
- Haskell predeclares some infix operators in the **standard prelude**, such as those for arithmetic.
- For each operator we need to specify its **precedence** and **associativity**. The higher precedence of an operator, the stronger it binds (attracts) its arguments: hence:

$$3 + 5 * 4 \equiv 3 + (5 * 4)$$

$$3 + 5 * 4 \not\equiv (3 + 5) * 4$$

## Declaring Infix Functions. . .

- The associativity of an operator describes how it binds when combined with operators of equal precedence. So, is

$$5-3+9 \quad \equiv \quad (5-3)+9 = 11$$

OR

$$5-3+9 \quad \equiv \quad 5-(3+9) = -7$$

The answer is that + and - associate to the **left**, i.e. parentheses are inserted from the left.

- Some operators are **right associative**:  $5^3^2 \equiv 5^{(3^2)}$
- Some operators have **free** (or **no**) associativity. Combining operators with free associativity is an error:

$$5 == 4 < 3 \quad \Rightarrow \quad \text{ERROR}$$

## Declaring Infix Functions...

- The syntax for declaring operators:

```
infixr prec oper  -- right assoc.
```

```
infixl prec oper  -- left assoc.
```

```
infix prec oper   -- free assoc.
```

\_\_\_\_\_ From the standard prelude: \_\_\_\_\_

```
infixl 7 *
```

```
infix 7 /, 'div', 'rem', 'mod'
```

```
infix 4 ==, /=, <, <=, >=, >
```

- An infix function can be used in a prefix function application, by including it in parenthesis. Example:

```
? (+) 5 ((* ) 6 4)
```

```
29
```

# Multi-Argument Functions

# Multi-Argument Functions

- Haskell only supports one-argument functions.
- An  $n$ -argument function  $f(a_1, \dots, a_n)$  is constructed in either of two ways:
  - 1 By making the one input argument to  $f$  a **tuple** holding the  $n$  arguments.
  - 2 By letting  $f$  “consume” one argument at a time. This is called **currying**.

Tuple	Currying
<code>add :: (Int,Int)-&gt;Int</code>	<code>add :: Int-&gt;Int-&gt;Int</code>
<code>add (a, b) = a + b</code>	<code>add a b = a + b</code>

# Currying

- Currying is the preferred way of constructing multi-argument functions.
- The main advantage of currying is that it allows us to define **specialized** versions of an existing function.
- A function is specialized by supplying values for one or more (but not all) of its arguments.
- Let's look at Haskell's plus operator (+). It has the type

`(+) :: Int -> (Int -> Int)`.

- If we give two arguments to (+) it will return an Int:

`(+) 5 3 ⇒ 8`



## Currying. . .

- If we just give one argument (5) to (+) it will instead return a **function** which “adds 5 to things”. The type of this specialized version of (+) is `Int -> Int`.

- Internally, Haskell constructs an intermediate – specialized – function:

```
add5 :: Int -> Int
add5 a = 5 + a
```

- Hence, `(+) 5 3` is evaluated in two steps. First `(+) 5` is evaluated. It returns a function which **adds 5 to its argument**. We apply the second argument `3` to this new function, and the result `8` is returned.

## Currying. . .

- To summarize, Haskell only supports one-argument functions. Multi-argument functions are constructed by successive application of arguments, one at a time.
- Currying is named after logician Haskell B. Curry (1900-1982) who popularized it. It was invented by Schönfinkel in 1924. **Schönfinkeling** doesn't sound too good...
- Note: Function application ( $f\ x$ ) has higher precedence (10) than any other operator. Example:

$$f\ 5\ +\ 1 \quad \Leftrightarrow \quad (f\ 5)\ +\ 1$$

$$f\ 5\ 6 \quad \Leftrightarrow \quad (f\ 5)\ 6$$

# Currying Example

- Let's see what happens when we evaluate `f 3 4 5`, where `f` is a 3-argument function that returns the sum of its arguments.

```
f :: Int -> (Int -> (Int -> Int))
```

```
f x y z = x + y + z
```

```
f 3 4 5 ≡ ((f 3) 4) 5
```

## Currying Example. . .

- $(f\ 3)$  returns a function  $f'$   $y\ z$  ( $f'$  is a specialization of  $f$ ) that adds 3 to its next two arguments.

$f\ 3\ 4\ 5 \equiv ((f\ 3)\ 4)\ 5 \Rightarrow (f'\ 4)\ 5$

$f' :: Int \rightarrow (Int \rightarrow Int)$

$f'\ y\ z = 3 + y + z$

## Currying Example. . .

- $(f' 4)$  ( $\equiv (f 3) 4$ ) returns a function  $f''$  ( $f''$  is a specialization of  $f'$ ) that adds  $(3+4)$  to its argument.

$$f 3 4 5 \equiv ((f 3) 4) 5 \Rightarrow (f' 4) 5 \\ \Rightarrow f'' 5$$

$$f'' :: \text{Int} \rightarrow \text{Int} \\ f'' z = 3 + 4 + z$$

- Finally, we can apply  $f''$  to the last argument (5) and get the result:

$$f 3 4 5 \equiv ((f 3) 4) 5 \Rightarrow (f' 4) 5 \\ \Rightarrow f'' 5 \Rightarrow 3+4+5 \Rightarrow 12$$

# Currying Example

\_\_\_\_\_ The Combinatorial Function: \_\_\_\_\_

- The combinatorial function  $\binom{n}{r}$  “n choose r”, computes the number of ways to pick  $r$  objects from  $n$ .

$$\binom{n}{r} = \frac{n!}{r! * (n - r)!}$$

\_\_\_\_\_ In Haskell: \_\_\_\_\_

```
comb :: Int -> Int -> Int
comb n r = fact n / (fact r * fact (n - r))
```

```
? comb 5 3
10
```

## Currying Example...

```
comb :: Int -> Int -> Int
comb n r = fact n / (fact r * fact (n-r))
```

```
comb 5 3 ⇒ (comb 5) 3 ⇒
           comb5 3 ⇒
           120 / (fact 3 * (fact 5-3)) ⇒
           120 / (6 * (fact 5-3)) ⇒
           120 / (6 * fact 2) ⇒
           120 / (6 * 2) ⇒
           120 / 12 ⇒
           10
```

```
comb5 r = 120 / (fact r * fact(5-r))
```

- comb<sup>5</sup> is the result of **partially applying** comb to its first argument.

# Associativity

- Function application is **left**-associative:

$$f\ a\ b = (f\ a)\ b \quad | \quad f\ a\ b \neq f\ (a\ b)$$

- The function space symbol ' $\rightarrow$ ' is **right**-associative:

$$a \rightarrow b \rightarrow c = a \rightarrow (b \rightarrow c)$$

$$a \rightarrow b \rightarrow c \neq (a \rightarrow b) \rightarrow c$$

- $f$  takes an `Int` as argument and returns a function of type `Int  $\rightarrow$  Int`.  $g$  takes a function of type `Int  $\rightarrow$  Int` as argument and returns an `Int`:

$f' :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$



$f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$



$g :: (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}$



# What's the Type, Mr. Wolf?

- If the type of a function  $f$  is

$$t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n \rightarrow t$$

- and  $f$  is applied to arguments

$$e_1 :: t_1, e_2 :: t_2, \dots, e_k :: t_k,$$

- and  $k \leq n$

- then the result type is given by cancelling the types  $t_1 \dots t_k$ :

$$\cancel{t_1} \rightarrow \cancel{t_2} \rightarrow \dots \rightarrow \cancel{t_k} \rightarrow t_{k+1} \rightarrow \dots \rightarrow t_n \rightarrow t$$

- Hence,  $f e_1 e_2 \dots e_k$  returns an object of type

$$t_{k+1} \rightarrow \dots \rightarrow t_n \rightarrow t.$$

- This is called the **Rule of Cancellation**.

# flip

```
flip      :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

- The `flip` function takes a function `f x y` (`f` is the function and `x` and `y` its two arguments), and reorders the arguments!
- Or, more correctly, `flip` returns a new function `f y x`.
- You can use this when you want to specialize a function by supplying an argument, but the function takes its arguments in the “wrong order.”

# flip...

- Consider the (!! ) function, for example:

```
> :type (!!)  
 (!! ) :: [a] -> Int -> a  
> :type flip (!!)  
 flip (!! ) :: Int -> [a] -> a  
> (!! ) [1..10] 2  
3  
> (flip (!! )) 2 [1..10]  
3
```

- Now you can write a function fifth using (!! ) which returns the fifth element of a list:

```
fifth :: [a] -> a  
fifth = (flip (!! )) 5
```

# Exercise

- Define an operator `$$` so that `x $$ xs` returns `True` if `x` is an element in `xs`, and `False` otherwise.

Example:

? `4 $$ [1,2,5,6,4,7]`

`True`

? `4 $$ [1,2,3,5]`

`False`

? `4 $$ []`

`False`

# Exercise

- Define an function `drop3` which takes a list as argument and returns a new list with the first three elements removed.
- Use currying!

# Exercise

```
> :type elem
elem :: Eq a => a -> [a] -> Bool
> elem 3 [1..10]
```

- The `elem` function returns true if the first argument is a member of the second (a list).
- Write a function `has3 xs` which returns true if `xs` (a list) contains the number 3.
- Write a function `isSmallPrime x` which returns true if `x` is one of the numbers 2,3,5,7.
- Use currying!

```
> isSmallPrime 2
True
> has3 [1]
False
```