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:
  - $3 + 5*4 \equiv 3 + (5*4)$
  - $3 + 5*4 \neq (3 + 5) * 4$

Declaring Infix Functions...

- The associativity of an operator describes how it binds when combined with operators of equal precedence. So, is $5-3+9 \equiv (5-3)+9 = 11$ OR $5-3+9 \equiv 5-(3+9) = -7$ 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:
  - $5 == 4 < 3$ \Rightarrow **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

- Haskell only supports one-argument functions.
- An $n$-argument function $f(a_1, \cdots, a_n)$ 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.

<table>
<thead>
<tr>
<th>Tuple</th>
<th>Currying</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add :: (Int, Int) -&gt; Int</code></td>
<td><code>add :: Int -&gt; Int -&gt; Int</code></td>
</tr>
<tr>
<td><code>add (a, b) = a + b</code></td>
<td><code>add a b = a + b</code></td>
</tr>
</tbody>
</table>

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 \textbf{Int $\rightarrow$ Int}.
- Internally, Haskell constructs an intermediate – specialized – function:
  \[
  \text{add5 :: Int} \rightarrow \text{Int} \\
  \text{add5 } a = 5 + a
  \]
- 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 Example

- Let’s see what happens when we evaluate \(f\ 3\ 4\ 5\), where \textbf{f} is a 3-argument function that returns the sum of its arguments.

\[
\begin{align*}
\text{f :: Int} & \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})) \\
\text{f } x \ y \ z & = x + y + z \\
\text{f } 3 \ 4 \ 5 & \equiv ((\text{f } 3) \ 4) \ 5
\end{align*}
\]

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. Schönfinkel doesn’t sound too good...
- Note: Function application \((f\ x)\) has higher precedence \((10)\) than any other operator. Example:

\[
\begin{align*}
\text{f } 5 & + 1 \quad \Leftrightarrow (\text{f } 5) + 1 \\
\text{f } 5 \ 6 & \quad \Leftrightarrow (\text{f } 5) \ 6
\end{align*}
\]

Currying Example...

- \((f\ 3)\) returns a function \textbf{f’} \ y \ z (\textbf{f’} is a specialization of \textbf{f}) that adds 3 to its next two arguments.

\[
\begin{align*}
\text{f } 3 \ 4 \ 5 & \equiv ((\text{f } 3) \ 4) \ 5 \Rightarrow (\text{f’ } 4) \ 5 \\
\text{f’ :: Int} & \rightarrow (\text{Int} \rightarrow \text{Int}) \\
\text{f’ } y \ z & = 3 + y + z
\end{align*}
\]
Currying Example

- \((f' \ 4) \ 5 = ((f \ 3) \ 4) \ 5 \Rightarrow (f' \ 4) \ 5 \Rightarrow f'' \ 5\)

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

  \[f \ 3 \ 4 \ 5 \Rightarrow (f \ 3) \ 4 \ 5 \Rightarrow (f' \ 4) \ 5 \Rightarrow f'' \ 5 \Rightarrow 3+4+5 \Rightarrow 12\]

Currying Example

- The combinatorial function \(\binom{n}{r}\) "\(n\) choose \(r\)" computes the number of ways to pick \(r\) objects from \(n\). \[\binom{n}{r} = \frac{n!}{r! \ast (n-r)!}\]

In Haskell:

\[\text{comb} :: \text{Int} \to \text{Int} \to \text{Int}\]
\[\text{comb} \ n \ r = \text{fact} \ n / (\text{fact} \ r \ast \text{fact}(n-r))\]

\[? \ \text{comb} \ 5 \ 3 \Rightarrow 10\]

Currying Example

- Function application is left-associative:
\[f \ a \ b = (f \ a) \ b \neq f (a \ b)\]

- The function space symbol `->` is right-associative:
\[a \to b \to c = a \to (b \to c) \neq (a \to b) \to c\]

- \(f\) takes an \(\text{Int}\) as argument and returns a function of type \(\text{Int} \to \text{Int}\). \(g\) takes a function of type \(\text{Int} \to \text{Int}\) as argument and returns an \(\text{Int}\):

\[f' :: \text{Int} \to (\text{Int} \to \text{Int}) \downarrow\]
\[f :: \text{Int} \to \text{Int} \downarrow\]
\[g :: (\text{Int} \to \text{Int}) \to \text{Int}\]

Associativity

- \(f g f = f (g f) \neq f g f\)
What’s the Type, Mr. Wolf?

- If the type of a function $f$ is
  $t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n \rightarrow t$
- and $f$ is applied to arguments
  $e_1::t_1, e_2::t_2, \cdots, e_k::t_k$,
- and $k \leq n$
- then the result type is given by cancelling the types $t_1 \cdots t_k$:

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

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

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

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

---

### flip

$$\text{flip} :: (a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c$$

- 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.”

---

### Homework

- 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
  ```

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

Use currying!