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:

\[
3 + 5*4 \equiv 3 + (5*4) \\
3 + 5*4 \not\equiv (3 + 5) * 4
\]

The syntax for declaring operators:
- \textbf{infixr} \texttt{prec oper} -- right associ.
- \textbf{infixl} \texttt{prec oper} -- left associ.
- \textbf{infix} \texttt{prec oper} -- free associ.

From the standard prelude:

\[
\text{infixr} \ 7 \ *\\
\text{infixl} \ 7 \ /, \textquoteright\text{div}, \text{\textquoteright}rem, \text{\textquoteright}mod\\n\text{infix} \ 4 \ ==, /=, \ <, \ <=, \ >, >=
\]

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

\[
? \ (+ \ 5 \ ((\ast) \ 6 \ 4)) \\
29
\]
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>\text{add} :: (\text{Int, Int}) \rightarrow \text{Int}</td>
<td>\text{add} :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})</td>
</tr>
<tr>
<td>\text{add} (a, b) = a + b</td>
<td>\text{add} \ a \ b = a + b</td>
</tr>
</tbody>
</table>

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

\[
(+) :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})
\]

- If we give two arguments to \((+).\) it will return an \text{Int}:

\[
(+) 5 3 \Rightarrow 8
\]

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 \text{Int} \rightarrow \text{Int}.

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

\[
\text{add5} :: \text{Int} \rightarrow \text{Int} \quad \text{add5} \ a = 5 + a
\]

- 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...

- 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:

\[
\begin{align*}
  f 5 + 1 & \equiv (f 5) + 1 \\
  f 5 6 & \equiv (f 5) 6
\end{align*}
\]
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.

\[
f :: \text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}))
\]

\[
f x y z = x + y + z
\]

\[
f \ 3 \ 4 \ 5 \equiv ((f \ 3) \ 4) \ 5
\]

Currying Example...

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

\[
f 3 4 5 \equiv ((f \ 3) \ 4) \ 5 \Rightarrow (f' \ 4) \ 5
\]

\[
f' :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})
\]

\[
f' y z = 3 + y + z
\]

Currying Example...

$(f' \ 4)$ $(\equiv (f \ 3) \ 4)$ returns a function $f'' \ z$ ($f''$ 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:

\[
\text{comb} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
\]

\[
\text{comb} n r = \text{fact} n / (\text{fact} r * \text{fact} (n-r))
\]

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

10
Currying Example...

comb :: Int -> Int -> Int

\[ \text{comb} n r = \frac{\text{fact } n}{\text{fact } r \times \text{fact } (n-r)} \]

\[ \text{comb} 5 3 \Rightarrow (\text{comb} 5) 3 \Rightarrow \]

\[ \text{comb}^5 3 \Rightarrow \]

\[ 120 / (\text{fact } 3 \times (\text{fact } 5-3)) \Rightarrow \]

\[ 120 / (6 \times (\text{fact } 5-3)) \Rightarrow \]

\[ 120 / (6 \times \text{fact } 2) \Rightarrow \]

\[ 120 / (6 \times 2) \Rightarrow \]

\[ 120 / 12 \Rightarrow \]

\[ 10 \]

\[ \text{comb}^5 r = 120 / (\text{fact } r \times \text{fact } (5-r)) \]

\[ \text{comb}^5 \] is the result of partially applying \text{comb} to its first argument.

Associativity

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

- The function space symbol \textit{\texttt{->}} is \textit{right}-associative:
  \[ a -> b -> c = a -> (b -> c) \]
  \[ a -> b -> c \neq (a -> b) -> c \]

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

\[ f' :: \texttt{Int -> (Int -> Int)} \]

\[ f :: \texttt{Int -> Int -> Int} \]

\[ g :: (\texttt{Int -> Int}) \rightarrow \texttt{Int} \]

What’s the Type, Mr. Wolf?

- If the type of a function \( f \) is\n  \[ t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t \]
  and \( f \) is applied to arguments\n  \[ e_1 :: t_1, e_2 :: t_2, \ldots, e_k :: t_k, \]
  and \( k \leq n \)
  then the result type is given by cancelling the types\n  \[ t_1 \ldots t_k \]
  \[ f_1 \rightarrow f_2 \rightarrow \ldots \rightarrow f_k \rightarrow t_{k+1} \rightarrow \ldots \rightarrow t_n \rightarrow t \]
  Hence, \( f \ e_1 \ e_2 \ldots e_k \) returns an object of type\n  \[ t_{k+1} \rightarrow \ldots \rightarrow t_n \rightarrow t \].
  This is called the \textit{Rule of Cancellation}.

Polymorphic Functions

- In Pascal we can’t write a \textit{generic} sort routine, i.e. one that can sort arrays of integers as well as arrays of reals:

\[ \text{procedure Sort (} \]
\[ \text{ var } A : \text{ array of } <\text{type}>; \]
\[ n : \text{ integer}); \]

- In Haskell (and many other FP languages) we can write \textit{polymorphic} (“many shapes”) functions.

- Functions of polymorphic type are defined by using \textit{type} variables in the signature:

\[ \text{length :: [a] -> Int} \]
\[ \text{length } s = \ldots \]
Polymorphic Functions...

- **length** is a function from lists of elements of some (unspecified) type `a`, to integer. I.e. it doesn't matter if we're taking the length of a list of integers or a list of reals or strings, the algorithm is the same.

  - `length [1,2,3]` ⇒ 3 (list of Int)
  - `length ["Hi ", "there", ", !"]` ⇒ 3 (list of String)
  - `length "Hi!"` ⇒ 3 (list of Char)

We have already used a number of polymorphic functions that are defined in the standard prelude.

- **head** is a function from “lists-of-things” to “things”:
  
  ```
  head :: [a] -> a
  ```

- **tail** is a function from lists of elements of some type , to a list of elements of the same type:
  
  ```
  tail :: [a] -> [a]
  ```

- **cons** "(:)" takes two arguments: an element of some type `a` and a list of elements of the same type. It returns a list of elements of type `a`:
  
  ```
  (:) :: a -> [a] -> [a]
  ```

Note that **head** and **tail** always take a list as their argument. **tail** always returns a list, but **head** can return any type of object, including a list.

- Note that it is because of Haskell's strong typing that we can only create lists of the same type of element. If we tried to do

  ```
  ? 5 : [True]
  ```

  the Haskell type checker would complain that we were consing an Int onto a list of Bools, while the type of "(:)" is

  ```
  (:) :: a -> [a] -> [a]
  ```

We want to define functions that are as reusable as possible.

1. **Polymorphic** functions are reusable because they can be applied to arguments of different types.
2. **Curried** functions are reusable because they can be specialized; i.e. from a curried function `f` we can create a new function `f'` simply by “plugging in” values for some of the arguments, and leaving others undefined.
A polymorphic function is defined using **type variables** in the signature. A type variable can represent an **arbitrary** type.

All occurrences of a particular type variable appearing in a type signature must represent the same type.

An identifier will be treated as an operator symbol if it is enclosed in backquotes: "'".

An operator symbol can be treated as an identifier by enclosing it in parenthesis: (+).

### Homework

Define a polymorphic function `dup x` which returns a tuple with the argument duplicated.

**Example:**

? dup 1
(1,1)

? dup "Hello, me again!"
("Hello, me again!",
"Hello, me again!")

? dup (dup 3.14)
((3.14,3.14), (3.14,3.14))

Define a polymorphic function `copy n x` which returns a list of `n` copies of `x`.

**Example:**

? copy 5 "five"
["five","five","five",
 "five","five"]

? copy 5 5
[5,5,5,5,5]

? copy 5 (dup 5)
[(5,5),(5,5),(5,5),(5,5),(5,5)]

Let `f` be a function from `Int` to `Int`, i.e. `f :: Int -> Int`. Define a function `total f x` so that `total f` is the function which at value `n` gives the total `f 0 + f 1 + \ldots + f n`.

**Example:**

```
double x = 2*x
pow2 x = x^2
totDub = total double
totPow = total pow2
? totDub 5
30
? totPow 5
55
```
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