1 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:

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

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

2 Declaring Infix Functions...

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

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

$$\text{OR}$$

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

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:

$$5 == 4 < 3 \Rightarrow \text{ERROR}$$
3 Declaring Infix Functions...

- The syntax for declaring operators:

  \texttt{infixr \textit{prec} \textit{oper} -- right assoc.}
  \texttt{infixl \textit{prec} \textit{oper} -- left assoc.}
  \texttt{infix \textit{prec} \textit{oper} -- free assoc.}

  \textbf{From the standard prelude:}

\texttt{infixl 7 *}
\texttt{infix 7 /, ‘div’, ‘rem’, ‘mod’}
\texttt{infix 4 ==, /=, <, <=, >=, >}

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

\texttt{? (+) 5 ((*) 6 4)}
\texttt{29}

4 Multi-Argument Functions

- Haskell only supports one-argument functions.

- An \( n \)-argument function \( f(a_1, \ldots, a_n) \) is constructed in either of two ways:
  \begin{enumerate}
  \item By making the one input argument to \( f \) a tuple holding the \( n \) arguments.
  \item By letting \( f \) “consume” one argument at a time. This is called \textit{currying}.
  \end{enumerate}

\begin{tabular}{l|l}
\textbf{Tuple} & \textbf{Currying} \\
\hline
\texttt{add :: (Int,Int)->Int} & \texttt{add :: Int->Int->Int} \\
\texttt{add (a, b) = a + b} & \texttt{add a b = a + b}
\end{tabular}

5 Currying

- Currying is the preferred way of constructing multi-argument functions.

- The main advantage of currying is that it allows us to define \textit{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

\texttt{(+) :: Int -> (Int -> Int)}.

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

\texttt{(+) 5 3 \Rightarrow 8}
6 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 \texttt{Int \rightarrow Int}.

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

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

- 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.

7 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:

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

8 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.

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

9 Currying Example...

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

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

10 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'' :: \ Int \rightarrow \ Int
f'' \ z = 3 + 4 + z

\bullet \ \text{Finally, we can apply } f'' \ \text{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

11 \ \text{Currying Example}

\textbf{The Combinatorial Function:}

\bullet \ \text{The combinatorial function } \binom{n}{r} \ \text{"n choose r", computes the number of ways to pick } r \ \text{objects from } n.

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

\text{In Haskell:}

\textbf{The Currying Example. . .}

\textbf{12 \ Currying Example. . .}

comb :: \ Int \rightarrow \ Int \rightarrow \ Int
comb \ n \ r = \text{fact } n / (\text{fact } r * \text{fact}(n-r))

? \comb \ 5 \ 3
10

13 \ \text{Associativity}

\bullet \ \text{Function application is left-associative: } f \ a \ b = (f \ a) \ b \ \text{if } f \ a \ b \neq f (a \ b)

\bullet \ \text{The function space symbol } '->' \ \text{is right-associative:}
a -> b -> c = a -> (b -> c)

• f takes an Int as argument and returns a function of type Int -> Int. g takes a function of type Int -> Int as argument and returns an Int:

\[
\begin{align*}
&f' :: \text{Int} \to \text{(Int} \to \text{Int)} \\
&f :: \text{Int} \to \text{Int} \to \text{Int} \\
&g :: \text{(Int} \to \text{Int}) \to \text{Int}
\end{align*}
\]

14 What’s the Type, Mr. Wolf?

• If the type of a function f is

\[
t_1 \to t_2 \to \cdots \to t_n \to t
\]

• and f is applied to arguments

\[
e_1 :: t_1, e_2 :: t_2, \ldots, e_k :: t_k,
\]

• and \( k \leq n \)

• then the result type is given by cancelling the types \( t_1 \cdots t_k \):

\[
f_1 \to f_2 \to \cdots \to f_k \to t_{k+1} \to \cdots \to t_n \to t
\]

• Hence, \( f \ e_1 \ e_2 \cdots e_k \) returns an object of type

\[
t_{k+1} \to \cdots \to t_n \to t.
\]

• This is called the Rule of Cancellation.

15 Polymorphic Functions

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

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

• In Haskell (and many other FP languages) we can write polymorphic (“many shapes”) functions.

• Functions of polymorphic type are defined by using type variables in the signature:

\[
\begin{align*}
&\text{length :: } [a] \to \text{Int} \\\n&\text{length } s = \ldots
\end{align*}
\]
16 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.

  \[
  \text{length } [1,2,3] \Rightarrow 3 \text{ (list of Int)} \\
  \text{length } ["Hi", "there", "!"] \Rightarrow 3 \text{ (list of String)} \\
  \text{length } "Hi!" \Rightarrow 3 \text{ (list of Char)}
  \]

17 Polymorphic Functions...

- 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”:

  \[
  \text{head} :: [a] \rightarrow a
  \]

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

  \[
  \text{tail} :: [a] \rightarrow [a]
  \]

- **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 \( a \):

  \[
  (:) :: a \rightarrow [a] \rightarrow [a]
  \]

18 Polymorphic Functions...

- 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

  \[
  (:) :: a \rightarrow [a] \rightarrow [a]
  \]

19 Summary

- 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.
20 Summary...

- 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: (+).

21 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))

22 Homework

- 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)]

23 Homework

- 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 + ... + f n`.

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

24 Homework

- 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