Comp 205:
Comparative Programming Languages

Higher-Order Functions

- Functions of functions
- Higher-order functions on lists
- Reuse

Lecture notes, exercises, etc., can be found at:
www.csc.liv.ac.uk/~grant/Teaching/COMP205/

Write a Haskell function `incAll` that adds 1 to each element in a list of numbers.

E.g., incAll [1, 3, 5, 9] = [2, 4, 6, 10]

```
incAll :: [Int] -> [Int]
incAll [] = []
incAll (n : ns) = n+1 : incAll ns
```

Write a Haskell function `lengths` that computes the lengths of each list in a list of lists.

E.g.,
```
lengths [[1,3], [1, [5, 9]]] = [2, 0, 2]
lengths ["but", "and", "if"] = [3, 3, 2]
```

```
lengths :: [[a]] -> [num]
lengths [] = []
lengths (l : ls)
    = (length l) : lengths ls
```

Write a Haskell function `map` that, given a function and a list (of appropriate types), applies the function to each element of the list.

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x : xs) = (f x) : map f xs
```

Using `map`

```
incAll = map (plus 1)
    where plus m n = m + n

lengths = map (length)
```

Note that `plus :: Int -> Int -> Int`, so `plus 1` :: Int -> Int.

Functions of this kind are sometimes referred to as partially evaluated.

Sections

Haskell distinguishes operators and functions: operators have infix notation (e.g. 1 + 2), while functions use prefix notation (e.g. `plus 1 2`).

Operators can be converted to functions by putting them in brackets: `(+) m n = m + n`.

Sections are partially evaluated operators. E.g.:

- `(+ m) n = m + n`
- `(0 :) 1 = 0 : 1`
Using map More

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>incAll</td>
<td>map (1 +)</td>
</tr>
<tr>
<td>addNewlines</td>
<td>map (++) &quot;\n&quot;</td>
</tr>
<tr>
<td>halveAll</td>
<td>map (/2)</td>
</tr>
<tr>
<td>squareAll</td>
<td>map (^2)</td>
</tr>
<tr>
<td>stringify</td>
<td>map (: [])</td>
</tr>
</tbody>
</table>

Write a Haskell function `product` that multiplies all the elements in a list of numbers together.

E.g., \[product\] \([2, 5, 26, 14]\) = 2*5*26*14 = 3640

```haskell
product :: [Int] -> Int
product [] = 1
product (n : ns) = n * product ns
```

Write a Haskell function `flatten` that takes a list of lists and appends all the lists together.

```haskell
flatten :: [[a]] -> [a]
flatten [] = []
flatten (l : ls) = l ++ flatten ls
```

A polynomial (in x) is an expression of the form: \(ax^3 + bx^2 + cx + d\)

The values a, b, c, d are called the coefficients. A polynomial can be represented as a list of the coefficients; e.g. \([1, 3, 5, 2, 1]\) represents the polynomial:

\[1x^4 + 3x^3 + 5x^2 + 2x + 1\]

Write a Haskell function

```haskell
poly :: Int -> [Int] -> Int
poly x [ ] = 1
poly x (a : as) = a * (x ^ length as) + poly (x ^ x : as)
```

Folding

A general pattern for the functions `product`, `flatten` and `polyLen` is replacing constructors with operators.

For example, `product` replaces `(cons)` with * and [] with 1:

```haskell
1 : (2 : (3 : (4 : [])))
```

We call such functions catamorphisms.
Folding Right

Haskell has a built-in function, `foldr`, that does this replacement:

\[
\text{foldr} : : (a \to b \to b) \to b \to [a] \to b
\]

\[
\text{foldr } f \ e \ [\] = e
\]

\[
\text{foldr } f \ e \ (x : xs) = f \ x \ (\text{foldr } f \ e \ xs)
\]

Using `foldr`

- `product = foldr (*) 1`
- `flatten = foldr (++) []`
- `poly x ns`
  - `p`
    - `where`
    - `(p,l) = polyLen ns`
    - `polyLen = foldr addp (0,0)`
    - `where`
    - `addp a (b,c)`
      - `= (a*(x^c)+b, c+1)`

Using `foldr` More

- `reverse : : [a] \to [a]`
  - `reverse = foldr swap []`
    - `where`
    - `swap h rl = rl ++ [h]`
  - `reverse (1 : (2 : []))`
    - `⇒ swap 1 (swap 2 [])`
    - `(swap 2 []) ++ [1]`
    - `([] ++ [2]) ++ [1]`

More Efficient Reversing

- `reverse = rev []`
  - `where`
  - `rev 1 [] = 1`
  - `rev 1 (x:xs) = rev (x:1) xs`
  - `rev [] (1 : (2 : []))`
    - `⇒ rev (1:[]) (2 : [])`
    - `⇒ rev (2:(1:[])) []`
    - `2 : (1 : [])`

Folding Left

Functions such as `rev` are called accumulators (because they accumulate the result in one of their arguments). They are sometimes more efficient, and can be written using Haskell’s built-in function `foldl`:

\[
\text{foldl} : : (b \to a \to b) \to b \to [a] \to b
\]

\[
\text{foldl } f \ e \ [\] = e
\]

\[
\text{foldl } f \ e \ (x : xs) = \text{foldl } f \ (f \ e \ x) \ xs
\]

Using `foldl`

- `rev = foldl swap []`
  - `where`
  - `swap acclist h = h : acclist`
- `flatten = foldl (++) []`
- `product = foldl (*) 1`
Compose

\[
\text{compose} :: \ (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c
\]

\[
\text{compose} \ f \ g \ x = f \ (g \ x)
\]

There is a Haskell operator \( \text{compose} \) that implements \( (f \ . \ g) \ x = f \ (g \ x) \)

Using Compose

\[
\text{poly} \ x = \text{fst} \ . \ (\text{foldr addp} \ (0,0))
\]

where

\[
\text{addp} \ a \ (p,1) = (a^*(x^1)+p, \ 1+1)
\]

\[
\text{makeLines} :: [[\text{char}]] \rightarrow [\text{char}]
\]

\[
\text{makeLines} = \text{flatten} \ . \ (\text{map} \ (++ \ "\n"))
\]

Summary

Functions are "first-class citizens" in Haskell: functions and operators can take functions as arguments.

Programming points:
- catamorphisms and accumulators
- sections

Next: user-defined types