

Functions on Tuples

- We usually use patterns to define functions on tuples
- For example, here is a function to add a pair of integers

```
addPair :: (Int, Int) -> Int
addPair (x, y) = x + y
```
- The pattern `(x, y)` matches any pair and sets `x` to be the first element of the pair and `y` to be the second element
- Example: `(0.0, a)` matches any pair where the first element is the `Float` value `0.0`; the second element of the pair is bound to `a`
- Example: `(0, ('5', answer))` matches any pair whose first element is `0` and whose second element is a pair whose first element is the character `'5'`; the second element of the second element is bound to `answer`

Lists as Compound Types

- A list is a sequence of elements of the same type, eg. `[1,2,3]` is of type `[Int]`, `["Mary","John","Sylwia"]` is of type `[String]`, etc.
- Every list is either empty, and it is then represented by `[]`, or non-empty, and then it can be written in the form `x:xs`
- `x` is the first item of the list and it is called *head*
- `xs` is the remainder of the list and it is called *tail*
- For example: `[1,2,3]` can be represented as `1:[2,3]`, where `1` is the head and `[2,3]` is the tail

Lists as Compound Types

- Every list can be built up repeatedly by applying the infix operator `:` (cons)
- `:` is a constructor for lists that makes a list by putting an element in front of an existing list:

`1:2:3: []`

`1:2: [3]`

`1: [2,3]`

`[1,2,3]`

- Note that here we use the fact that `:` is *right associative*, so that for any values of `x`, `y` and `zs`

`x:y:zs = x:(y:zs)`

List Functions

- Suppose we wanted to define a function to add together all elements of a list of integers (there is a standard function `sum` that does this, but ignore this for the sake of the exercise)

```
sumList :: [Int] -> Int
```

- For a fixed length it is easy:

```
sumList3 [x, y, z] = x + y + z
```

- But how about a function `sumList` that would work for *any length*?

List Functions

- Just as we described calculating factorials, we can think of laying out the values of `sumList` as follows

```
sumList [] = 0
```

```
sumList [8] = 8
```

```
sumList [37,8] = 45
```

```
sumList [9,37,8] = 54
```

```
sumList [1,9,37,8] = 55
```

- And just as in the case of factorial, we can describe the whole process by describing the first line and how to go from one line to the next as follows:
 - the sum of the empty list is 0
 - the sum of a non-empty list `x:xs` is `x + sum of xs`

List Functions

- The inductive design of `sumList` translates directly into the following Haskell definition:

```
sumList :: [Int] -> Int
sumList [] = 0
sumList (x:xs) = x + sumList xs
```

- This gives a definition of `sumList` by *primitive recursion over lists*
- In such a definition we give
 - a starting point: the value of `sumList` at `[]`, and
 - a way of going from the value of `sumList` at a particular point – `sumList xs` – to the value of `sumList` at the next point, namely `sumList (x:xs)`

List Functions

```
sumList :: [Int] -> Int
sumList [] = 0
sumList (x:xs) = x + sumList xs
```

- Notice how patterns are used to distinguish the empty and non-empty cases
- Notice how the second clause defines `sumList` in terms of `sumList` applied to a "smaller" argument
- The argument in the recursive calls gets smaller and smaller until it is the empty list and hence the recursion terminates through the first clause, *ie.* the base case
- Notice how the definition is like "mathematical" induction: the base case is a list of length 0 and the inductive case defines the function for lists of length $n+1$ in terms of the result for lists of length n

List Functions

- Consider the evaluation of `sumList [1,9,37,8]`
- Using the equation `sumList (x:xs) = x + sumList xs` repeatedly we have:
`sumList [1,9,37,8]`
 $\Rightarrow 1 + \text{sumList } [9,37,8]$
 $\Rightarrow 1 + (9 + \text{sumList } [37,8])$
 $\Rightarrow 1 + (9 + (37 + \text{sumList } [8]))$
 $\Rightarrow 1 + (9 + (37 + (8 + \text{sumList } [])))$
- And then using the equation `sumList [] = 0` and integer arithmetic we get:
 $\Rightarrow 1 + (9 + (37 + (8 + 0)))$
 $\Rightarrow 55$
- The recursion used to define `sumList` will give an answer on any finite list since each recursion steps takes us closer to the base case where `sumList` is applied to `[]`

List Functions

- Example: `length` is a standard function but how would we define it ourselves?
- Design:
 - base case: the length of `[]` is 0
 - inductive case: a non-empty list is 1 element longer than its tail

```
length :: [a] -> [Int]
length [] = 0
length (x:xs) = 1 + length xs
```
- Try it yourself: evaluate the following expression
`length [1,9,37,8]`

List Functions

- Example: a function to double every element of a list of integers

```
double [1,9,37,8]
```

```
⇒ [2,18,74,16]
```

- Design:

- base case: doubling all elements of [] returns []

- inductive case: multiply the head of the list by 2 and use : (cons) to put it in the front of double of the tail of the list

```
double :: [Int] -> [Int]
```

```
double [] = []
```

```
double (x:xs) = (2 * x) : double xs
```

- Try it yourself: evaluate the following expression

```
double [1,9,37]
```

List Functions

- Example: how would we define the (standard) list append function (`++`) that joins two lists together?

`[2,3,4] ++ [9,8]`

\Rightarrow `[2,3,4,9,8]`

- This is trickier to design because (`++`) takes 2 lists
- Should we try induction on the first list, the second list or both?
- For recursion on both lists we have the following 4 cases, the combination of base and inductive cases for each list:

`[] ++ [] = ...`

`[] ++ (y:ys) = ...`

`(x:xs) ++ [] = ...`

`(x:xs) ++ (y:ys) = ...`

- If we start with this collection, we may find some redundancies and eliminate them

List Functions

- Examples: taking 2 for x and $[3,4]$ for xs we have
 $[2,3,4] ++ [9,8] \Rightarrow [2,3,4,9,8]$
 $[3,4] ++ [9,8] \Rightarrow [3,4,9,8]$
- So we get $[2,3,4] ++ [9,8]$ by putting 2 on the front of $[3,4] ++ [9,8]$
- In the case that the first list is empty:
 $[] ++ [9,8] \Rightarrow [9,8]$
- We finally get the following definition for $(++)$:
 $(++) :: [a] \rightarrow [a] \rightarrow [a]$
 $[] ++ ys = ys$
 $(x:xs) ++ ys = x:(xs++ys)$

List Functions

- So in fact, recursion on just the first list is sufficient to define this particular function
- The first two cases do not need to be separated
 $[] ++ [] = \dots$
 $[] ++ (y:ys) = \dots$
are covered by a single clause $[] ++ ys = ys$
- The third and fourth cases
 $(x:xs) ++ [] = \dots$
 $(x:xs) ++ (y:ys) = \dots$
are also combined into $(x:xs) ++ ys = x:(xs++ys)$
- Attempting to define $(++)$ by induction on only the second list does not seem possible:
 $xs ++ [] = xs$
 $xs ++ (y:ys) = ?$
- The problem is that the basic list constructor $:$ adds elements to the *front* of a list; the pattern $(y:ys)$ binds y to some element that will end up in the *middle* of the result list...

List Functions

- Designing a function that sorts a list of integers into ascending order:

```
iSort :: [Int] -> [Int]
```

- For example: `iSort [7,3,9,2] ⇒ [2,3,7,9]`

- As in all previous examples, our first intuition should be the two cases:

```
iSort [] = ...
```

```
iSort (x:xs) = ...
```

- The first case is easy since the empty list is trivially sorted:

```
iSort [] = []
```

List Functions

- For non-empty lists we have direct access to the head x and the tail xs , *ie.* $7 : [3, 9, 2]$
- Typically, there is a recursive call of the function on the tail of the list
- Hence we might expect the second clause to look like:
$$\text{iSort } (x:xs) = \dots (\text{iSort } xs) \dots$$
- For the example above, we have x equal to 7 and $\text{iSort } xs$ equal to $[2, 3, 9]$
- To build the final result, $\text{iSort } (x:xs)$ from x and $\text{iSort } xs$, we need to insert 7 into the correct position in $[2, 3, 9]$
- Hence we can write:
$$\text{iSort } (x:xs) = \text{insert } x (\text{iSort } xs)$$

Top-Down Design

- This is a very easy example of *top-down design*
- `iSort` has been defined assuming that we can define `insert`
- Top-down design involves breaking a problem into smaller and smaller sub-problems until they become manageable or trivial
- How we go about this process of factoring into smaller parts is the *essence* of program design

Abstraction

- The most basic skill in designing and constructing programmes in any language is *abstraction*
- All we need to know to understand and design `iSort` is *what* the `insert` function does
- The details of *how* `insert` works is irrelevant to the definition above
- At one stage we may produce a solution to some problem while ignoring *how* the subcomponents work; we are only interested in *what* they do
- Then we may concentrate *separately* on *how* each subcomponent is to work, while ignoring *why* the overall design needs them
- Having factored the overall problem into smaller sub-problems, we must assess whether the decomposition gave us *simpler* sub-components; if not, we will re-visit the overall design
- Programme design and development is an *iterative process*

List Functions

- We have written `iSort`, using a function `insert` which we must now define

- Again our intuition should be to try the standard inductive approach:

```
insert :: Int -> [Int] -> [Int]
insert x [] = ...
insert x (y:ys) = ...
```

- To insert `x` into an empty list maintaining ascending ordering is trivial:

```
insert x [] = [x]
```

- If the list is non-empty, we have direct access to its head; we may ask: does `x` belong before or after the head of the list?

- if `x` belongs before the head, simply cons it there and we are done
- if `x` belongs after the head, we can use a tail-recursive call of `insert` to find out where it belongs:

```
insert x (y:ys)
  | x <= y = x:y:ys
  | otherwise = y:insert x ys
```

- Thus the definition of `iSort` gives:

```
iSort :: [Int] -> [Int]
```

```
iSort [] = []
```

```
iSort (x:xs) = insert x (iSort xs)
```

```
insert :: Int -> [Int] -> [Int]
```

```
insert x [] = [x]
```

```
insert x (y:ys)
```

```
    | x <= y = x:y:ys
```

```
    | otherwise = y:insert x ys
```

- Try it yourself: evaluate `iSort [3,9,2]`

List Functions

- Pattern matching is a readable and powerful way of defining list functions in Haskell
- The pattern restricts the set of values to which an equation applies
- Patterns *are not* arbitrary expressions but literal values (such as 0, [], etc.) or constructed values (such as (n+1), (x:xs), etc.) and hence `sqrt (x^2) = x` is *illegal*
 - the pattern [] matches an empty list
 - the pattern (x:xs) matches a non-empty list, binding x to the head of the list and xs to the tail of the list
- Variables *cannot* be repeated on the left-hand side of a definition: `findReps (x:x:xs) = x:findReps xs` is *illegal* and so is `find code ((code,name,price):rest) = (name,price)`

List Functions

- Define a function that sums the pairs of elements in a list:

```
sumPairs :: [(Int,Int)] -> [Int]
```

```
sumPairs [(1,9),(37,8)]
```

```
⇒ [10,45]
```

- The base case is standard: `sumPairs [] = []`
- We can choose a variety of patterns for the inductive case
- Alternatives:

```
sumPairs1 (p:ps) = (fst p + snd p):sumPairs1 ps
```

```
sumPairs2 (p:ps) = (m+n):sumPairs2 ps  
    where (m,n) = p
```

```
sumPairs ((m,n):ps) = (m+n):sumPairs ps
```

- In the final alternative, the most precise pattern has been used, leading to the simplest and clearest definition of the function

List Functions

- Haskell attempts to match patterns in a strictly *sequential order*:
 - *left to right* in a definition and
 - *top to bottom* in a sequence of definition clauses
- Note that as soon as a pattern matches *fails*, that equation is rejected and the next one tried
- Consider:

```
myZip :: [a] -> [b] -> [(a,b)]
myZip (x:xs) (y:ys) = (x,y):myZip xs ys
myZip xs ys = []
```
- If the first argument is [] then the match on (x:xs) fails and that clause is rejected *without* attempting to match the second argument; `myZip` truncates the longer of the two list arguments