

- 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 usual for functions over lists, the first intuition should be the two cases:

```
iSort [] = ... (basic case)
```

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

- The first case is easy since the empty list is trivially sorted:
 $\text{iSort } [] = []$
- For non-empty lists we have direct access to the head x and the tail xs
- Typically, there is a recursive call of the function on the tail of the list, and hence we might expect the second clause to look like:
 $\text{iSort } (x:xs) = \dots (\text{iSort } xs) \dots$

- For example, to sort `[7,3,9,2]`, we need to insert the head 7 in the right place into the sorted tail `[3,9,2]`:
 - 1 sort the tail: `[2,3,9]`
 - 2 inset the head 7 in the right place in this list: `[2,3,7,9]`
- Hence based on our previous pattern
`iSort (x:xs) = ... (iSort xs) ...`
we can write the following recursive clause to define `iSort`:
`iSort (x:xs) = insert x (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 of `iSort`
- 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*

Program design

- 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 \Rightarrow simply put it there (`:`) and that is it
- if `x` belongs after the head \Rightarrow 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]`

Development Cycle

We can see programs being developed in a cycle:

- Understand the problem
- Design the solution
- Write the solution
- Evaluate, test

Understanding the Problem

- To do this properly (especially in the context of lab/exam sessions), the first step is to read entirely the instructions (specifications) provided and to observe carefully the function call examples and returned results
- What are the inputs and outputs to the problem? Are there any special conditions on the inputs or outputs?
- Does the problem itself have a structure? Is it made up of parts which could be solved separately? (cf. top-down design)

Understanding the Problem

`myProduct :: [Int] -> Int` multiplies the elements of a list of integers

- The input is a list of integers `[Int]` while the output is a single integer `Int`
- What if the input list is empty?
 - The output must be an element of type `Int`, and hence we cannot write: `myProduct [] = []`
 - The output value resulting from the recursive function application must not be changed when the basic case is reached, and hence we cannot write: `myProduct [] = 0` since this would set the output value to 0
 - Solution: `myProduct [] = 1`

Understanding the Problem

`sumPoly :: Poly -> Poly -> Poly` sums two polynomials

- The inputs are two lists of integers (type synonym `Poly`) and the output is a single list of integers
 - A list is to be built as output
 - Use the cons operator (`:`) in the recursive step to build the final list
- What are the different cases the function definition must account for given the lists may contain a different number of elements?
 - Both lists are empty: `sumPoly [] [] = []`
 - Either the first or the second list is empty and in this case the returned list is the non-empty input list itself:
`sumPoly xs [] = xs`
`sumPoly [] ys = ys`
 - Both lists are non-empty:
`sumPoly (x:xs) (y:ys) = (x+y) : sumPoly xs ys`

Understanding the Problem

`select :: Ord a => a -> [a] -> [a]` extracts items whose value is less than a given threshold `a` from a list `[a]`

```
select y [] = []
```

```
select y (x:xs)
```

```
  | x < y = x:select y xs
```

```
  | otherwise = select y xs
```

- Based on `select` define a function, say `iCount`, which returns the number of times a given value `y` occurs in a list `xs`
- What has to be changed?
 - The function implements a *count* \Rightarrow the output is no longer a list but an `Int` (cf. *number of times*):
`iCount :: Ord a => a -> [a] -> Int`
 - Basic case \Rightarrow the output has to be an `Int`:
`iCount y [] = 0`
 - `x` and `y` have to be checked for equality (`==`) and not for less than (`<`) and `1` is to be added as the result in this case:
`| x == y = 1 + iCount y xs`

Understanding the Problem

- To extract items whose value is less than a given threshold a from a list $[a]$

```
select :: Ord a => a -> [a] -> [a]
select y [] = []
select y (x:xs)
  | x < y = x : select y xs
  | otherwise = select y xs
```

- To count the number of times a given value a occurs in a list $[a]$

```
iCount :: Ord a => a -> [a] -> Int
iCount y [] = 0
iCount y (x:xs)
  | x == y = 1 + iCount y xs
  | otherwise = iCount y xs
```

Designing a Solution

- Have you seen a similar problem before? If so, you might use its design as a guide
- Can you think of a simpler but related problem? If you can solve that, you might use or modify the solution
- Make sure you know what your resources are, *ie.* predefined functions and programs you yourself have already written

Designing a Solution

- Identify the objects
- Identify the basic operations on objects
- Choose representations of the objects
- Design the basic operations on the objects
- Factor the process into manageable parts

Writing a Program

- Make sure you know Haskell's syntax
 - type names start with a capital letter
 - user-defined functions' names start with a lower-cased letter
 - indentation is used to spread function definitions across several lines
 - a list is represented as $(x:xs)$
 - etc.
- Recursion is a general strategy for designing programmes over data types like numbers and lists
- The operator `:` makes it possible to build lists; use it in the recursive clause each time you need to build a list as output
- The recursive function call must follow the pattern specified in the function declaration and used on the left-hand side of the function definition
 - `evalPoly a (x:xs) = x + a * (evalPoly xs)` *is not correct*
 - `evalPoly a (x:xs) = x + a * (evalPoly a xs)` *is correct*
- Work some examples by hand

Evaluating a Program

- Can you test your solution?
- You *always* need to check if it works on general cases as well as on hard and special cases
- Use the function calls and results provided as examples; try on your own examples
- If the testing reveals errors or bugs, try to find their source (cf. Hugs error messages and line of the script that led to their generation)

Designing a Solution

- Identify the objects
- Identify the basic operations on objects
- Choose representations of the objects
- Design the basic operations on the objects
- Factor the process into manageable parts

Case Study – Problem and Objects

- Problem: find all the misspelt words in a text file; report as a lexically ordered list
- Objects
 - dictionary against which to check words from the file
 - the result of the check is a list of words
 - the input is a file path name for the text file
 - the contents of the text file will be a sequence of characters (*ie.* a string) to be interpreted as a sequence fo words
- Hence the objects are: dictionay, file path name, word, list of words

Case Study – Basic Operations on Objects

- Get file contents
- Extract words from file contents
- Look up word in dictionary
- Compare words for equality
- Sort list of words

Case Study – Representations of the Objects

- Words can be represented as strings:
`type Word = String`
- The file path name will also be a string, as defined in the prelude:
`type FilePath = String`
- The contents of the file will also be a string, to be broken into a sequence (*ie.* a list) of words:
`type Words = [Word]`
- How should we represent the dictionary? The simplest representation is a list of words:
`type Dictionary = Words`

Case Study – Factor Process into Manageable Parts

- Input: file path
- Output: ordered list of the words that do not appear in the dictionary
- Four main steps seem appropriate:
 - 1 get the contents of the input file
 - 2 get the words from the file
 - 3 check if each word is in the dictionary
 - 4 sort the misspelt words into alphabetic order

file path	⇒	<code>readFile</code>	⇒	contents
contents	⇒	<code>wordsFromString</code>	⇒	list of words
list of words	⇒	<code>misspelt</code>	⇒	misspelt words
misspelt words	⇒	<code>sort</code>	⇒	alphabetical list

Case Study – Overall Programme

- The overall programme will be the composition of functions corresponding to those steps:

```
wordsFromString :: String -> Words
```

```
misspelt :: Words -> Words
```

- Note that the dictionary is fixed, not an argument:

```
sort :: Ord a => [a] -> [a]
```

- The sequence is:

```
check :: String -> Words
```

```
check contents = sort (misspelt (wordsFromString  
contents))
```

- Or equivalently:

```
check = sort . misspelt . wordsFromString
```

- `.` is an operator used for *function composition* in which the output of one function (eg. `sort`) becomes the input of another (eg. `misspelt`)

Case Study – Implement Basic Operations

- Look up a word in the dictionary:
Since the dictionary is a simple list of words, we can use for example the standard function `elem` (`elem :: Int -> [Int] -> Int`) which checks whether an `Int` is an element of `[Int]` extending it to the type `String`
- Compare words for equality:
The equality operator (`==`) applies to strings
- Sort list of words:
We have developed an insertion sort function which can be used since strings are an ordered type
- Get file contents:
The standard function is `readFile`

Case Study – Implement Basic Operations

- Extract words from file contents:
This is more complicated and requires to be decomposed into smaller parts
- Words are lists of alphabetic characters so:
 - 1 skip over non-letters in the file contents `String` (`skipToLetter`)
 - 2 the first word now appears at the beginning of the string (`firstWord`)
 - 3 repetitively apply those steps to the remainder of the string (`restOfString`) until there are no more words left

Case Study – Implement Basic Operations

- Some standard functions are useful:

`isAlpha :: Char -> Bool` tests whether characters are letters

`takeWhile :: (a -> Bool) -> [a] -> [a]` returns the longest initial segment of `xs` whose elements all satisfy the property `p`; a property `p` over a type `a` is represented by a function of type `(a -> Bool)`

For example: `takeWhile even [2,4,6,1,5,6] ⇒ [2,4,6]`

`dropWhile :: (a -> Bool) -> [a] -> [a]` returns the remaining portion of the list

For example: `dropWhile even [2,4,6,1,5,6] ⇒ [1,5,6]`

- We have:

`skipToLetter = dropWhile (not . isAlpha) s`

`firstWord = takeWhile isAlpha skipToLetter`

`restOfString = dropWhile isAlpha skipToLetter`

Case Study – Implement Basic Operations

- Put them together to return a sequence of words:

```
wordsFromString :: String -> Words
```

```
wordsFromString s
```

```
  | firstWord == '' = []
```

```
  | otherwise = firstWord : wordsFromString  
                    restOfString
```

```
where
```

```
skipToLetter = dropWhile (not . isAlpha) s
```

```
firstWord = takeWhile isAlpha skipToLetter
```

```
restOfString = dropWhile isAlpha skipToLetter
```