9 Program Design II — Case Study

Aim: Design and develop a Haskell program to check for spelling errors.

Recall the steps in program design:

1: Understand the Problem

2: Identify the Objects

3: Identify the Basic Operations on Objects

4: Choose Representations of the Objects

5: Implement the Basic Operations on Objects

6: Factor the Process into Manageable Parts

1: Understand the Problem

More detail: Find all the misspelt words in a text file. Report as a lexically ordered list.

The complete program is available in `Scripts/labw7/Spell.lhs` or via the CE1 web page.

2: Identify the Objects

Clearly we need a *dictionary* against which to check words from the file.

The result of the check is a *list of words*.

The input will be a *file path name*.

The *contents of the file* will be a sequence of characters (i.e. a string) to be interpreted as a sequence of *words*.

Objects:

dictionary
file path name
word
list of words

3: Identify the Basic Operations on Objects

- look up word in dictionary
- compare words for equality
- sort list of words
- get file contents
- extract words from file contents

4: Choose 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 of words:

  ```
  > type Words = [Word]
  ```

- How should we represent a dictionary?

We will choose the simplest representation:

```
type Dictionary = Words
```

because this is a prototype program and we have not yet studied better alternatives.

6: Factor the Process into Manageable Parts

The input is a file path. The output is an ordered list of the words that do not appear in the dictionary.

Three main steps seem appropriate:

1. Get the contents of the file
2. Get the words from (the contents of) the file
3. Check if each word is in the dictionary
4. Sort the misspelt words into alphabetic order

file path	$\Rightarrow$	**readFile**	$\Rightarrow$	contents
contents	$\Rightarrow$	**wordsFromString**	$\Rightarrow$	list of words
list of words	$\Rightarrow$	**misspelt**	$\Rightarrow$	misspelt words
misspelt words	$\Rightarrow$	**sort**	$\Rightarrow$	alphabetical list

The overall program will be the *composition* of functions corresponding to those steps.

Ignoring the I/O for now, we have:

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

Note that the dictionary is fixed, not an argument.

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

The composition (or sequence) is:

```
> check :: String -> Words
> check contents =
>     sort (misspelt (wordsFromString contents))
```

or equivalently:

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

To complete, add the I/O:

```
> spellCheck :: FilePath -> IO()
> spellCheck f
>   = do{contents <- readFile f;
>        putStr (unlines (check contents))}
```

5: Implement the Basic Operations on Objects

Look up a word in the dictionary:
Since the dictionary is a simple list of words, we can just use the standard functions `elem` or `notElem`.

Compare words for equality:
The equality operator `(==)` applies to strings.

Sort list of words:
We have developed an insertion sort function. It can be used since strings are an ordered type.

Get file contents:
Use the standard function `readFile`.

Extract words from file contents:
This is more complicated.

Decompose this process into smaller parts (as in step 5).

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.

Some standard functions are useful.

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

`dropWhile :: (a -> Bool) -> [a] -> [a]`
`takeWhile :: (a -> Bool) -> [a] -> [a]`
`takeWhile p xs` returns the longest initial segment of `xs` whose elements all satisfy predicate `p`. `dropWhile p xs` returns the remaining portion of the list.

We have:

```
> skipToLetter  = dropWhile (not . isAlpha) s
> firstWord     = takeWhile isAlpha skipToLetter
> restOfString  = dropWhile isAlpha skipToLetter
```

Putting them together to return the 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
```