COMP3141

Software System Design and Implementation

Practical 1

Zoltan A. Kocsis
University of New South Wales
Term 2 2022

Staff

I am Zoltan A. Kocsis (Research Asst., CSE): I'm a researcher at the Trustworthy Systems group at UNSW. I work on the applications of formal mathematical methods to the development of safe and secure software.

Raphael Douglas Giles (CSE): will deliver many of these practical lectures (Thursdays).

James Davidson (Casual Academic, Math): course admin, course forums.

Word Frequencies

Let's solve a problem to get some practice implementing stuff:

Example (Task 1)

Given a number n and a string s containing English words, generate a report that lists the n most common words in the given string s.

I'll even give you an algorithm:

1. Break the input string into words.
2. Convert the words to lowercase.
3. Sort the words.
4. Group adjacent occurrences (runs) of the same word.
5. Sort runs words by length.
6. Take the longest n runs of the sorted list.
7. Generate a report.

Demo: word frequencies, using Hoogle

The Dollar Pattern

We used *the dollar operator* $ to reduce the use of parentheses.

- The dollar operator does normal function application, like `f x` (evaluation of a function at a value).
- However, while application has high operator precedence ("is done as early as possible"), the dollar operator has extremely low precedence ("is done as late as possible").
- `reverse [1,2,3] ++ [4]` results in `[3,2,1,4]`. The application of the `reverse` function binds very tightly, so we do it first, then concatenate.
- `reverse $ [1,2,3] ++ [4]` results in `[4,3,2,1]`. We concatenate first, then apply the reversing function. Same as `reverse ([1,2,3] ++ [4])`.

Function Composition

We used *function composition* to combine our functions together. The mathematical $(f \circ g)(x)$ is written `(f . g) x` in Haskell.

In Haskell, operators like function composition are themselves functions. You can define your own!

```
-- Vector addition
(.+) :: (Int, Int) -> (Int, Int) -> (Int, Int)
(x1, y1) .+ (x2, y2) = (x1 + x2, y1 + y2)
```

```
(2,3) .+ (1,1) == (3,4)
```

You could even have defined function composition yourself if it didn't already exist:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

Conditionals

Demo: polarity using guards, if statements.

Demo: loops via recursion.

Lists

We used a bunch of list functions. How could we implement them ourselves??

Lists are singly-linked lists in Haskell. The empty list is written as `[]` and a list node is written as `x : xs`. The value `x` is called the head and the rest of the list `xs` is called the tail. Thus:

```
"hi!" == ['h', 'i', '!'] == 'h':('i':('!':[]))
                         == 'h' : 'i' : '!' : []
```

When we define recursive functions on lists, we use the last form for pattern matching:

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

Equational Evaluation

```
map f []     = []
map f (x:xs) = f x : map f xs
```

We can evaluate programs *equationally*:

```
map toUpper "hi!"  ≡  map toUpper ('h':"i!")
                   ≡  toUpper 'h' :  map toUpper "i!"
                   ≡  'H' : map toUpper "i!"
                   ≡  'H' : map toUpper ('i':"!")
                   ≡  'H' : toUpper 'i' : map toUpper "!"
                   ≡  'H' : 'I' : map toUpper "!"
                   ≡  'H' : 'I' : map toUpper ('!':"")
                   ≡  'H' : 'I' : '!'  :  map toUpper ""
                   ≡  'H' : 'I' : '!'  :  map toUpper []
                   ≡  'H' : 'I' : '!'  :  []
                   ≡  "HI!"
```

FIN

The quiz is available online. You can start working on it, but you cannot submit it yet due to technical issues. There will be an announcement once these are resolved.

Warning

The quiz is assessed. The deadline is the end of next Thursday.

If the technical difficulties take too long to resolve, you'll be given an extension. Watch out for the announcement.