School of Computing and Information Systems
COMP30026 Models of Computation Tutorial Week 1

24–30 Jul. 2023

Introduction to Tutorials

Welcome to the first Models of Computation tutorial for 2023. We hope you will enjoy the tutorials, and also that you will contribute to making them enjoyable for your class mates.

Your tutor will explain how they would like to run the tutorials. Let your tutor know who you want to work with and they will be happy to oblige. Your willingness to engage is critical and we expect you to be active in tutorials.

The Optional section is completely optional. There is no expectation to go through them in the tutorial.

Exercises

T1.1 In Lecture 1 we play a word rewriting game. Here is an example of a similar kind of game, the so-called Post's Correspondence Problem. We have a set of "dominoes" such as

$$\left\{\boxed{\frac{\texttt{b}}{\texttt{ca}}}, \boxed{\frac{\texttt{a}}{\texttt{ab}}}, \boxed{\frac{\texttt{ca}}{\texttt{a}}}, \boxed{\frac{\texttt{abc}}{\texttt{c}}}\right\}$$

The dominoes use a small alphabet, say the letters `a`, `b`, and `c`. Each domino has a string written in the upper half, and one in the lower half. The given set of dominoes is always finite, but there is an unbounded supply of each domino, that is, we can use it many times. The goal is to find a sequence of dominoes which, when laid side by side, spell out identical non-empty strings across the top and the bottom (or alternatively, determine that no solution is possible). For example, the set given above has a solution, namely

$$\boxed{\frac{\texttt{a}}{\texttt{ab}}}\boxed{\frac{\texttt{b}}{\texttt{ca}}}\boxed{\frac{\texttt{ca}}{\texttt{a}}}\boxed{\frac{\texttt{a}}{\texttt{ab}}}\boxed{\frac{\texttt{abc}}{\texttt{c}}}$$

(a) Can you find a solution to $\left\{\boxed{\frac{\texttt{baa}}{\texttt{abaaa}}}, \boxed{\frac{\texttt{aaa}}{\texttt{aa}}}\right\}$?

(b) How about $\left\{\boxed{\frac{\texttt{a}}{\texttt{cb}}}, \boxed{\frac{\texttt{bc}}{\texttt{ba}}}, \boxed{\frac{\texttt{c}}{\texttt{aa}}}, \boxed{\frac{\texttt{abc}}{\texttt{c}}}\right\}$?

(c) How about $\left\{\boxed{\frac{\texttt{ab}}{\texttt{aba}}}, \boxed{\frac{\texttt{bba}}{\texttt{aa}}}, \boxed{\frac{\texttt{aba}}{\texttt{bab}}}\right\}$?

T1.2 Can you see the general algorithm that solves Post's Correspondence Problem?

T1.4 Visit Grok and work through Haskell Module 1, "Functions and Recursion". If you are fast, you may even get started on Module 2 on lists. Support the other students in your group.

(We don't plan to use tutorial time on Grok from Week 2 onwards; this exercise is just to make sure you get started.)

T1.5 Consider an n by m grid. We want to find out how many different paths there are that one can travel along, starting from the bottom left and ending in the top right cell, given that *one can only move up or right*. Write a Haskell function

```
paths :: Integer -> Integer -> Integer
```

so that `paths m n` returns the number of such paths. (We use `Integer`s as the numbers can get quite large.)

Optional

T1.6 (Optional; wordy; interesting if you love functions and/or have been racing ahead.)

Module 1 pointed out that Haskell programmers don't like the mathematicians' convention of putting parentheses around function arguments. Instead of $f(x)$ we write `f x`. Juxtaposing `a` and `b` means "apply `a` to `b`" (so `a` must be a function). Function application is written as juxtaposition. Instead of $a(b, c, d)$ we write `a b c d`.

To understand how this can work, first note that Haskell has truly *higher-order* functions: Any function can appear as an argument to another function, and, moreover, it can be the result produced by a function.

When you write `a b c`, Haskell reads that as `(a b) c`. That is, `a` applied to `b`, and the result of that is applied to `c`. So the way Haskell thinks of `a` is that it is a function which produces a function—it is higher-order. Applied to `b`, it gives a *function* (`a b`) which is then applied to `c`.

Note that `a b c` is therefore different to `a (b c)`. The latter tells us that both `a` and `b` are functions (and not necessarily higher-order), because `b` is applied to `c` and then `a` is applied to the result.

So, we write `a b c d` instead of $a(b, c, d)$. But to be honest, we can actually write it as `a (b,c,d)` in Haskell as well. Haskell programmers don't tend to do that, and there is a clear difference: With `a b c d`, the function `a` takes an argument `b` and produces a higher-order function (which takes and argument `c` and produces a function ...) With `a (b,c,d)`, `a` again takes one argument, but this time it is a *triple*, and `a` is not (necessarily) higher-order. The two variants of `a` have different *types*. We'll get to types in Module 3, but for now let us just say that the higher-order `a` has a type of the form $\beta \to \gamma \to \delta \to \rho$, while the other version has a type of the form $(\beta, \gamma, \delta) \to \rho$.

It turns out that, for every function f of type $\alpha \to \beta \to \gamma$, there is a corresponding h of type $(\alpha, \beta) \to \gamma$, that is, satisfying $f\ x\ y = h(x, y)$ for all x, y, and vice versa. We say that f is the *curried* version of h and h is the *uncurried* version of f. Here we assumed two arguments, so h takes a pair, but the correspondence holds for three arguments (triples), four, etc.

(a) Let g be the curried version of $h : (\alpha, \beta, \gamma) \to \rho$. Wanting to use `g`, the curried version, how does the Haskell programmer write $h(x, y, f(x))$?

(b) Write a Haskell function `curry2` which takes a function of type $(\alpha, \beta) \to \gamma$ and returns its curried version.

(c) Write a Haskell function `uncurry2` which takes a function of type $\alpha \to \beta \to \gamma$ and returns its uncurried version.

(d) None of the versions is spicier than the other. We "curry" a function in honour of the logician H. B. Curry. Find Curry's first name.

T1.7 (Optional, time consuming.)

Write a Haskell function `a2r` to translate a positive integer to the corresponding roman numeral, using the symbols M, D, C, L, X, V, and I. The meaning of these: M=1000, D=500, C=100, L=50, X=10, V=5, and I=1. For example, `a2r 2009` should yield the string `"MMIX"`. If you find more time, write a function to translate the other way too. For example, `r2a "DXXVI"` should yield 526. If you have way too much time on your hand, add a function which checks that a string is a well-formed roman numeral.

Why we still use Roman numerals is a mystery—they have no redeeming features. The value of a string of symbols is the sum of the symbols' values, except: when a smaller value appears

before a larger, the contribution of that pair is the difference between the larger and the smaller. The rules for well-formedness are rather cumbersome:

(a) I can precede V and X, but not L, C, D, or M.

(b) X can precede L or C, but not D or M.

(c) C can precede D or M.

(d) V, L, or D can't be followed by a numeral of greater or equal value.

(e) A symbol with two consecutive occurrences can't be followed by a symbol of larger value.