Discussion 3 FAQs

1 Assignment 2 - Lambda Calculus

Q: How do I go about designing a function?

There are a bunch of ways to think about this below is an example of coming up with the successor function.

Take the successor function as an example (I will use slightly different parameter names than in the assignment). We want $Succ\ n_\lambda = (n+1)_\lambda$. We know we will want to have a number as our first parameter so we create a lambda abstraction that will take in a number.

$$Succ\ = \lambda n....$$

Now we know we want to return a number. A number is just a function that takes in two parameters, so we can add two more lambda abstractions.

$$Succ\ = \lambda n.\lambda x.\lambda y....$$

Since a number is just the first parameter, (x), applied n times to y we need to apply x to our y one extra time. First we apply x to y n times using our number:

$$Succ\ = \lambda n.\lambda x.\lambda y.nxy$$

At this point we have recreated the same number with some indirection. The x and y we take into our function will be directly passed to the number. Currently $Succ\ n_\lambda = n_\lambda$. We need to add one more application of x to increase the number:

$$Succ\ = \lambda n.\lambda x.\lambda y.x(nxy)$$

Now we have the extra application of x and we get $Succ\ n_\lambda = (n+1)_\lambda$. Note in the homework the n parameter is labelled u but the functions are the same.

2 Assignment 3 - ML

Q: When do you need or not need parenthesis in ML?

ML binds similarly to the lambda calculus, that is, function application binds to the left.

```
- fun add x y = x + y;
val add = fn : int -> int -> int
- add 1 2;
val it = 3 : int
```

In the above the expression `add 1 2` can be read as `(add 1) 2`. Where `(add 1)` is a function that takes in a number and returns a new function.

Infix operators bind *looser* than function application. `add 1 2 * add 3 4` would be read as `(add 1 2) * (add 3 4)`. The parenthesis don't hurt so feel free to add them in if it makes the expression more clear.

Q: How do I read the type annotations in ML?

Being able to read type annotations is helpful for knowing if you've written your code correctly. Take the type annotation for `map`:

```
val map = fn : ('a -> 'b) -> 'a list -> 'b list
               |---+----|    |--+--|    |--+--|
                   |            |          |
                   |        a list of      |          a list of
                   |        elements of    +--------- elements of
          function from     generic type              generic type
          generic type 'a   'a                        'b
          to generic type
          'b
```

This makes sense because we know the first parameter to `map` is a function that operators on each element of the list and `('a -> 'b)` is a function type from one generic type (`'a`) to another (`'b`). Now we would expect that we need to take in a list of `'a`s and we will apply the function to each element to produce a list of `'b`s. The second underlined type is the list of `'a`s that the function takes in as input, while the last underlined type is the output type, the list of `'b`s.

You could read this type in two ways: "A function that takes in a function and a list and returns a transformed list" **or** " A function that takes in a function and returns another function from a list to a transformed list":

```
val map = fn : ('a -> 'b) -> 'a list -> 'b list
                             |-------+---------|
                                     |
                               A function that takes in a
                               list of 'a and returns a list
                               of 'b                  ^
```

```
                                                      |
               |-------------+-----------------|      |
                             |                        |
                             |                        |
                    A function that takes in a        |
                    a function from 'a to 'b          |
                    and returns ----------------------+
```

Q: How would you get a particular index of a list?

You can write a “get” function:

```
(* Define an exception for the error case *)
exception IndexOutOfBounds;

(*    index
        |
        | list
        |  |       *)
fun get n nil     = raise IndexOutOfBounds (* If we ran out list *)
  | get 0 (h::t) = h                      (* location found *)
  | get n (h::t) = get (n - 1) t;         (* recur on the tail *)

(*
    The type of get is:

    get : int -> 'a list -> 'a
           |     |--+--|     |
        index       |        |
                  list       |
                           element at index
*)
```

Here we are travelling the list `n` times until we reach the appropriate element.