# COMP3400
# Assignment 1 Written

Paul Vrbik

March 3, 2023

## Lambda Calculus

We can encode boolean logic in lambda calculus as follows:

$$\mathsf{True} = \lambda x.(\lambda y.x) \qquad \mathsf{False} = \lambda x.(\lambda y.y)$$
$$\mathsf{And} = \lambda p.(\lambda q.(pq)p) \qquad \mathsf{Or} = \lambda p.(\lambda q.(pp)q)$$

### Question 1. [1 MARK]

Give the $\lambda$-expression for NOT that takes True to False and vice-versa. Your solution should be in its $\beta$-normal form.

```
Not = \a.(a (False) (True))
= \a.(a (\x(\y.y)) (\x(\y.x))) ------STOP
--------
= a[a:= (\x(\y.y)] (\x(\y.x))
= (\x(\y.y)) (\a(\b.a))
= (\y.y)[x:=(\a(\b.a))]
=\y.y
```

### Question 2. [5 MARKS]

Recall that $\neg(p \wedge q) \equiv \neg p \vee \neg q$ and thereby Or is redundant because

$$p \vee q \equiv \neg(\neg p \wedge \neg q).$$

Give the $\lambda$-expression for $\neg(\neg p \wedge \neg q)$ and show it is equivalent to Or.

### Question 3. [4 MARKS]

Reduce the following lambda expression to its $\beta$-normal form.

$$(\lambda xy.x)(\lambda abc.cab)z(\lambda z.zz).$$

```
= (\xy.x)[x:=\abc.cab]z(\z.zz)
= (\y.(\abc.cab))z(\z.zz)
= ((\abc.cab))[y:=z](\z.zz)
= (\abc.cab)(\z.zz)
= (\bc.cab)[a:=\z.zz]
= \bc.(c(\z.zz)b)
= \bc.(c(zz)[z:=b])
= \bc.(cbb)
```

# Principal Types

There is no partial credit for this section. You are not allowed to use `undefined`.

The answers for these questions can be checked automatically by Haskell so this "written" work will be submitted to the autograder via the `PrincipalType.hs` file.

The questions are given in ascending difficulty.

## Question 4. [2 MARKS]

Define a function `f1` such that

```
> :type f1
f1 :: (a -> b, a) -> b
```

up to renaming of the type variables. Your function does not have to be total but should *not* be undefined.

## Question 5. [2 MARKS]

Same instructions as Question 4 but with

```
f2 :: a -> (b, c) -> b
```

## Question 6. [2 MARKS]

Same instructions as Question 4 but with

```
f3 :: (a -> a) -> a -> [a]
```

Note. There are several ways to implement this function but we want most general one that produces the most meaningful result. Trivial implementations like `f3 _ _ = []` will not be accepted since they are not general and do not produce anything meaningful.

## Question 7. [2 MARKS]

Same instructions as Question 4 but with

```
f4 :: (b -> r) -> (a -> b) -> (a -> r)
```

f4 f4-1 f4-2 = f4-1 f4-2

## Question 8. [1 MARK]

Same instructions as Question 4 but with

```
f5 :: ((a, b, c) -> d) -> a -> b -> c -> d
```

abc

f5 abc a b c = abc a b c

## Question 9. [1 MARK]

Same instructions as Question 4 but with

```
f5_inv ::  (a -> b -> c -> d) -> (a, b, c) -> d
```

## Principal Types: Extra

This is not part of the assignment, but rather some context which makes the previous page of questions more meaningful.

### Question 4

*Curry–Howard isomorphism* or *equivalence* is the direct relationship between computer programs and mathematical proofs. It states that if there is a total program with a specific type, then the logical statement corresponding to that type is true. The function

```
f1 :: (a -> b, a) -> b
```

represents Modus Ponens. It states that

> "If the statement '$A$ implies $B$' is true and statement $A$ is true, then statement $B$ is also true."

In the function type, `a -> b` corresponds to the statement $A \Rightarrow B$ ($A$ implies $B$) and `a` corresponds to statement $A$.

By implementing this function, you will prove Modus Ponens.

### Question 8 and Question 9

These functions are another example of Curry-Howard isomorphism. By implementing them you will prove the powers law:

$$((d^a)^b)^c = d^{abc}.$$

If there are $m$ elements in the set $A$ and $n$ elements in the set $B$, then the number of functions from $A$ to $B$ (with type $A \to B$) is $n^m$.

`f5` receives a function of type

```
((a, b, c) -> d)
```

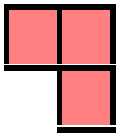
and returns a function

```
(a -> b -> c -> d)
```

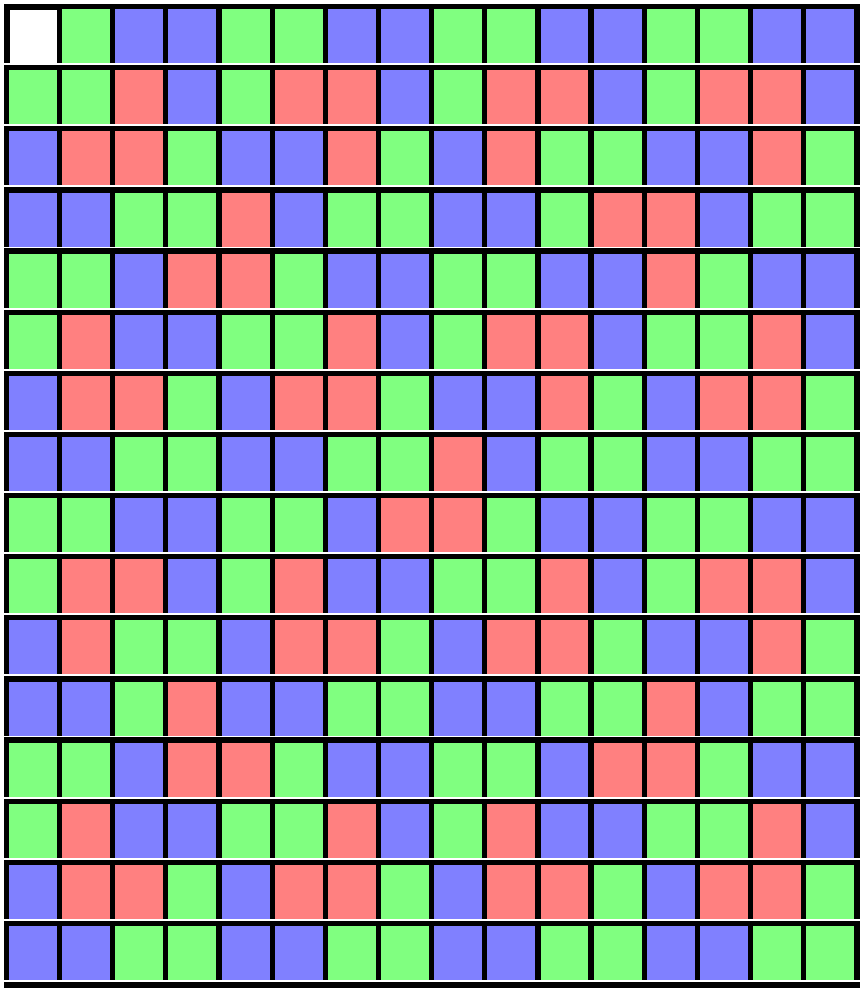
The number of functions of type `(a, b, c) -> d` is $d^{abc}$ and the number of functions of type `a -> b -> c -> d` is $((d^a)^b)^c$.

# Blockus

There is a game called Blockus where players try and fill a grid of squares with Tetris-like pieces. One such piece is called "V3" and looks like...



If we *remove a corner square* from a $16 \times 16$ board we can cover what remains with V3 pieces.



## Question 10. [10 MARKS]

Most of the programs we write in Haskell will be *recursive* or *inductive* in nature. The purpose of this question is to help us get into the mindset of reasoning inductively.

Use the *principle of mathematical induction* to prove a $2^n \times 2^n$ Blockus board with north-west corner removed can be covered with V3 pieces.

*Note*: This question will be marked very thoroughly. We will be looking for the presence of all necessary components of induction to be *stated clearly*. You will be marked down for being unnecessarily verbose or for making unsubstantiated claims. Every statement you write should be clearly inferred from the statements that precede it (not statements that come after).

Essentially we are looking for *clear* and *concise* proofs.