What to turn in

Turn in the assignment during class on the due date.

1. Proofs by induction (25 pts.)
   Consider a version of IMP that has **for** loops instead of **while** loops. We redefine commands c as follows:

   $$c ::= \textbf{skip} \mid x := a \mid \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \mid c_0;\ c_1 \mid \textbf{for } x = a_0 \textbf{ to } a_1 \textbf{ do } c$$

   Let us suppose that the large-step semantics are unchanged except that we substitute the following **for** rules for the **while** rules:

   $$\frac{\langle a_0, \sigma\rangle \Downarrow n_0 \quad \langle a_1, \sigma\rangle \Downarrow n_1}{\langle \textbf{for } x = a_0 \textbf{ to } a_1 \textbf{ do } c, \sigma\rangle \Downarrow \sigma} \text{ where } n_0 > n_1$$

   $$\frac{\langle a_0, \sigma\rangle \Downarrow n_0 \quad \langle a_1, \sigma\rangle \Downarrow n_1 \quad \langle \textbf{for } x = n_0 \textbf{ to } n_1 \textbf{ do } c, \sigma\rangle \Downarrow \sigma'}{\langle \textbf{for } x = a_0 \textbf{ to } a_1 \textbf{ do } c, \sigma\rangle \Downarrow \sigma'} \text{ where } n_0 \leq n_1$$

   $$\frac{\langle c;\ \textbf{for } x = n_0' \textbf{ to } n_1 \textbf{ do } c, \sigma[x \mapsto n_0]\rangle \Downarrow \sigma'}{\langle \textbf{for } x = n_0 \textbf{ to } n_1 \textbf{ do } c, \sigma\rangle \Downarrow \sigma'} \text{ where } n_0 \leq n_1 \wedge n_0' = n_0 + 1$$

   Informally, the bounds of the loop are computed once, at the beginning of the loop, and although the loop index variable can be assigned within the loop, these assignments do not affect the value of the variable at the beginning of the next loop iteration.

   (a) Write a program in this language that, given an input number in the variable n, outputs the nth prime number in the variable x.

   (b) Define a series of programs $P_1, P_2, P_3, \ldots$ such that the length of program P_n is polynomial in n but the running times of the programs grow faster than any exponential in n.

   (c) Despite the fact that we can write many useful programs in this language—it can compute the *primitive recursive functions*—the language is not universal. Show that it is not universal by demonstrating that all programs terminate. (*Hint:* Use well-founded induction, but make sure you show your well-founded relation is indeed well-founded!)

2. Free and bound variables (10 pts.)
   Identify the free and bound variables in each of the following expressions:

   (a) $(\lambda(\mathsf{x\ y\ z})\ \mathsf{z\ x\ y})$

   (b) $(\lambda(\mathsf{x\ y})\ (\lambda\mathsf{z}\ (\mathsf{z\ y}))\ (\lambda\mathsf{x}\ (\mathsf{z\ x})))$

   (c) $((\lambda(\mathsf{x\ y})\ \mathsf{y})\ \mathsf{x})$

   We defined substitution into a lambda term using the following three rules:

   $$\begin{aligned}
   (\lambda x\ e_0)\{e_1/x\} &= (\lambda x\ e_0) \\
   (\lambda y\ e_0)\{e_1/x\} &= (\lambda y\ e_0\{e_1/x\}) \qquad (\text{where } y \neq x \wedge y \notin FV[\![e_1]\!]) \\
   (\lambda y\ e_0)\{e_1/x\} &= (\lambda y'\ e_0\{y'/y\}\{e_1/x\}) \qquad (\text{where } y' \neq x \wedge y' \notin FV[\![e_0]\!] \wedge y' \notin FV[\![e_1]\!])
   \end{aligned}$$

   (d) In these rules, we note a number of conjuncts in the side-conditions whose purpose is perhaps not immediately apparent. Show by counterexample that each of the conjuncts above is independently necessary.

3. Fixed Point Combinator (10 pts.)
In class we have seen a fixed point operator Y, which has the property $Yf = f(Yf)$. This is not the only fixed point operator. Prove that the combinator B defined as the following is also a fixed pointer operator.

$A \equiv \lambda(\mathsf{a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ o\ p\ q\ s\ t\ u\ v\ w\ x\ y\ z\ r})\ \mathsf{r(t\ h\ i\ s\ i\ s\ a\ f\ i\ x\ e\ d\ p\ o\ i\ n\ t\ c\ o\ m\ b\ i\ n\ a\ t\ o\ r)}$
$B \equiv \mathsf{A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A}$

4. Encodings (25 pts.)
We have seen in class one way to represent natural numbers in the λ-calculus. However, there are many other ways to encode numbers in λ-calculus. Consider the following definitions:

$$\mathsf{TRUE} \equiv \lambda(\mathsf{x\ y})\ \mathsf{x}$$
$$\mathsf{FALSE} \equiv \lambda(\mathsf{x\ y})\ \mathsf{y}$$
$$0 \equiv \lambda\mathsf{x}\ \mathsf{x}$$
$$n+1 \equiv \lambda\mathsf{x}\ (\mathsf{x\ FALSE})\ n$$

(a) Show how to write the DEC (decrement by one) operation for this number representation. Reduce (DEC (DEC 2)) to its $\beta\eta$ normal form, which should be the representation of 0, above.

(b) Show how to write a λ-term ZERO? that determines whether a number is zero or not. It should return TRUE when the number is zero, and FALSE otherwise. Use the definitions of TRUE and FALSE given above.

(c) Show how to write the ADD and MULT operations for this number representation.

5. The S and K Combinators (30 pts.)
Consider the following definitions of the S and K combinators:

$$S \equiv \lambda(\mathsf{x\ y\ z})\ ((\mathsf{x\ z})\ (\mathsf{y\ z}))$$
$$K \equiv \lambda(\mathsf{x\ y})\ \mathsf{x}$$

Any λ-calculus expression without free variables can be written using only applications of the S and K combinators; thus, the λ-calculus can be universal with only three distinct identifier names, since both combinators use no more than three identifiers.

(a) Show that the S and K combinators can be used to construct an expression with the same normal form as the identity expression $I \equiv \lambda\mathsf{x}\ \mathsf{x}$.

(b) Now, we will construct a translation from λ-calculus expressions to expressions containing only applications of the S and K combinators. This translation will be defined in terms of two functions: $\mathcal{C}[\![e]\!]$, which converts an expression e into this form, and a function $\mathcal{A}[\![x, e]\!]$, which *abstracts* the variable x from the expression e. removing all uses of x within e.
The idea is that $\mathcal{A}[\![x, e]\!] = \lambda x\ e$, in the sense that the two expressions have the same effect when applied to any argument (they are extensionally equal). Using the function $\mathcal{A}$, the function $\mathcal{C}$ can be defined simply by structural induction:

$$\begin{aligned} \mathcal{C}[\![x]\!] &= x \\ \mathcal{C}[\![e_0\ e_1]\!] &= (\mathcal{C}[\![e_0]\!]\ \mathcal{C}[\![e_1]\!]) \\ \mathcal{C}[\![\lambda x\ e]\!] &= \mathcal{A}[\![x, \mathcal{C}[\![e]\!]]\!] \end{aligned}$$

Because $\mathcal{A}$ is only applied to expressions produced by $\mathcal{C}$, it needs to be defined only for expressions that are identifiers and applications. For example, consider $\mathcal{A}[\![x, x']\!]$ where $x' \neq x$. We require $(\mathcal{A}[\![x, x']\!]e) = (\lambda x\ x')e$ for any e, so we obtain the right effect with the following definition:

$$\mathcal{A}[\![x, x']\!] = (K\ x') \qquad \text{(where } x \neq x')$$

Define the remainder of the translation to the S and K combinators. Does this translation result in the most compact equivalent expression using these combinators? Justify your answer.