What to turn in

Turn in the written parts of the assignment during class on the due date.

Problem 1: REC Language

(a) Prove the Substitution Lemma, which shows that substituting x with e' in e yields the same denotation as evaluating e using an extended environment which maps x to the denotation of e':

$$\mathcal{C}[\![e]\!]_h \phi\rho[x \to \mathcal{C}[\![e']\!]_h \phi\rho] = \mathcal{C}[\![e[x \to e']]\!]_h \phi\rho$$

(b) Consider the following recursive function declaration in REC with call-by-value semantics:

```
f(X) = ifnz X then f(f(X-2)) else 0
```

1. Write the equations that define the function environment for this recursive function;
2. Write the function environment solution and show that it is the least fixed point for the above equations.

Problem 2: Normal Form

Define the length of a combinator in lambda calculus as the number of variable occurrences. For instance, the length of $\lambda x.\, x(\lambda y.\, xyx)$ is 4.

First, we explore the difference in number of steps for the evaluation of lambda terms under call-by-name and call-by-value semantics, using β reductions only. In some evaluations, CBV semantics may be more efficient (i.e., evaluates to normal form in fewer steps) than CBN semantics.

(a) Write a combinator of length $O(n)$, whose evaluation takes exactly n steps (β reductions) under CBV semantics, but takes $2n+c$ steps under CBN semantics, where c is a constant. Show how the evaluation proceeds and explain why the evaluations have the required lengths.

(b) Same as part (a), but with CBV evaluation in $O(n)$ and CBN evaluation in $\Omega(2^n)$ steps.

We next explore the growth of lambda expressions in the process of reducing to the normal form, under different evaluations. For each of the questions, you must show how the reduction works and why the normal forms meet the requirements.

(c) Write a combinator of length $O(n)$, whose normal form with CBN evaluation is of length $\Omega(2^n)$.

(d) Write a combinator of length $O(n)$, whose normal form with full β reduction evaluation is of length $\Omega(n^n)$;

Note: the O and Ω notations refer to the order of growth of functions: f is $O(g)$ if f grows slower than a constant factor of g and f is $\Omega(g)$ if f grows faster than a constant factor of g.

Problem 3: Lambda Calculus Representations

We want to use a representation of integers different than Church numerals. As computer scientists (read "hackers"), we feel more comfortable with a binary representation of numbers. Let's then try to encode numbers as follows:

$$\begin{array}{rcl}
\overline{0} & = & \lambda i.\, \lambda o.\, \lambda t.\, ot \\
\overline{1} & = & \lambda i.\, \lambda o.\, \lambda t.\, it \\
\overline{2} & = & \lambda i.\, \lambda o.\, \lambda t.\, i(ot) \\
\overline{3} & = & \lambda i.\, \lambda o.\, \lambda t.\, i(it) \\
\overline{4} & = & \lambda i.\, \lambda o.\, \lambda t.\, i(o(ot)) \\
\overline{5} & = & \lambda i.\, \lambda o.\, \lambda t.\, i(o(it)) \\
\overline{6} & = & \lambda i.\, \lambda o.\, \lambda t.\, i(i(ot)) \\
\overline{7} & = & \lambda i.\, \lambda o.\, \lambda t.\, i(i(it)) \\
\overline{8} & = & \lambda i.\, \lambda o.\, \lambda t.\, i(o(o(ot))) \\
\ldots & &
\end{array}$$

Intuitively, we represent number n as a function which takes three parameters, i as the bit 1, o as the bit 0, and t as the sequence terminator, and returns the sequence of applications representing the binary encoding of n. Your job is to derive combinators for manipulating such numbers.

(a) Write a combinator *testz* for testing if a number is zero;

(b) Write a combinator for multiplying a number by two.

(c) Write two combinators *tolist* and *fromlist* for converting such numbers to and from lists of booleans, where true elements in the list represent one bits and false elements represent zero bits;

(d) Write a combinator *add* for adding two binary numbers;

(e) Write a combinator *log* for computing $\lfloor \log_2 n \rfloor$. The result must be a binary number;

(f) Since Church numerals don't handle negative numbers, we will try to represent such numbers by extending the above binary representation. The new representation of number n is still a function of the form $\lambda i.\, \lambda o.\, e$, but the expression e represents now the sign-extended binary representation of n. Assume that we represent signed numbers using 4 bits.

Write a combinator *twoc* for computing the two's complement of a signed binary number and then use it to define the signed addition function *sadd* for such numbers.