What to turn in

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

Problem 1: REC Language

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

```
f(X) = ifz X then 0 else g(X-2)
g(X) = ifz X then 0 else f(X-3)
```

a) Write the equations that define the function environment for this recursive function;

b) 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.

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

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

For each question you must show how the reduction works and why the normal forms meet the requirements.

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: Fixed Point Combinators

Show the term B defined below is a fixed point combinator:

$$\begin{array}{rcl} A & = & \lambda(e\,a\,i\,s\,v\,x\,l\,n).\, n(s\,i\,x\,e\,l\,e\,v\,e\,n) \\ B & = & AAAAAAAA \end{array}$$

Problem 4: The S and K Combinators

Consider the following two combinators:

$$\begin{array}{rcl} S & \equiv & \lambda x.\, \lambda y.\, \lambda z.\, x\, z\, (y\, z) \\ K & \equiv & \lambda x.\, \lambda y.\, x \end{array}$$

It turns out that any closed term in the λ-calculus can be expressed in terms of these two combinators.

a) Show how to express $I = \lambda x.\, x$ in terms of the S and K combinators.

We want to write a translation of every lambda term into a combinator consisting only of S and K, without any variable or abstraction except those occurring in S and K.

For each term e (not necessarily closed), we want to define a translation $\mathcal{T}[\![e]\!]$, which is a lambda term $\beta\eta$-equivalent to e, and which consists only of applications of S, K, and free variables. The resulting term contains exactly the same free variables as the original expression. The grammar that describes the expressions generated by this translation is:

$$e ::= x \mid e_0 e_1 \mid S \mid K$$

We want to define the translation $\mathcal{T}[\![e]\!]$ inductively, as follows:

$$\begin{array}{rcl} \mathcal{T}[\![x]\!] & = & x \\ \mathcal{T}[\![e_0 e_1]\!] & = & \mathcal{T}[\![e_0]\!]\mathcal{T}[\![e_1]\!] \\ \mathcal{T}[\![\lambda x.\, e_0]\!] & = & \mathcal{L}[\![x, \mathcal{T}[\![e_0]\!]]\!] \end{array}$$

where $\mathcal{L}[\![x, e]\!]$ is a helper translation which abstracts away the variable x in e and replaces it with an expression consisting only of S and K. The resulting expression should be equivalent to $\lambda x.\, e$.

b) For what terms e do we need to define the helper translation $\mathcal{L}[\![x, e]\!]$?

c) Define the translation $\mathcal{L}[\![x, e]\!]$ for the terms mentioned above.

d) Formally show that: $\mathcal{T}[\![e]\!] =_{\beta\eta} e$. Here we assume that α-equivalence holds by default.

e) Does your translation for closed terms (i.e., combinators) result in the most compact expression using only S and K? Justify your answer.

Problem 5: 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:

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

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.

(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 for multiplying two numbers.