What to turn in

Turn in the written parts of the assignment during class on the due date. For the programming part, you should mail your version of the file `translate.ml` to `nystrom@cs.cornell.edu` by 5PM on that day. The short problems (4,5) will not be worth as much as the others.

1. Substitution Lemma

   We define the small step parallel reduction semantics for lambda expressions as follows:

$$\frac{}{x \mapsto_p x}$$

$$\frac{t_1 \mapsto_p t_3 \quad t_2 \mapsto_p t_4}{t_1\ t_2 \mapsto_p t_3\ t_4}$$

$$\frac{t_1 \mapsto_p t_2}{\lambda\ x\ t_1 \mapsto_p \lambda\ x\ t_2}$$

$$\frac{t_1 \mapsto_p t_3 \quad t_2 \mapsto_p t_4}{(\lambda\ x\ t_1)\ t_2 \mapsto_p t_3\ \{t_4/x\}}$$

   Prove that $(t_1 \mapsto_p t_2)\ \wedge\ (t_3 \mapsto_p t_4) \Rightarrow t_1\{t_3/x\} \mapsto_p t_2\{t_4/x\}$.

2. Evaluation Contexts

   We extend the call-by-value λ calculus with a primitive exception mechanism as follows:

$$e ::= \ x \mid \lambda\ x\ e \mid e_1\ e_2 \mid \mathsf{raise} \mid \mathsf{try}\ e_1\ \mathsf{handle}\ e_2$$

   The expression **raise**, encountered in either the operand or operator position of an application, raises an exception and terminates the current evaluation of the expression. The exception will be bubbled up until a **try** expression is reached. If e_1 in the **try** expression raises an exception, we will start to evaluate e_2 in the exception handler in the next step.

   raise is not a value in this language. For example, try $((\lambda\ x\ 1)$ raise) handle 2 evaluates to 2 instead of 1, because $((\lambda\ x\ 1)$ raise) evaluates to raise instead of 1.

   Evaluation contexts C for this language can be defined as:

$$\begin{aligned} v &::= \ \lambda\ x\ e \\ C &::= \ [\,] \mid C\ e \mid v\ C \mid \mathsf{try}\ C\ \mathsf{handle}\ e \end{aligned}$$

   A redex r_a for this evaluation context can have the following forms:

$$r_a ::= (\lambda\ x\ e)\ v \mid \mathsf{try}\ D\ \mathsf{handle}\ e \mid \mathsf{try}\ v\ \mathsf{handle}\ e$$

   Here, D is an expression that contains an "exposed" **raise**: a **raise** that is not hidden in an λ abstraction or a **try** expression. The formal definition of D is:

$$D ::= \ \mathsf{raise} \mid D\ e \mid v\ D$$

Given these definitions, the possible reductions are as follows:

$$(\lambda\ x\ e)\ v \mapsto_a e\ \{v/x\}$$
$$\mathsf{try}\ D\ \mathsf{handle}\ e\ \mapsto_a e$$
$$\mathsf{try}\ v\ \mathsf{handle}\ e\ \mapsto_a v$$

And we have the usual rule for reductions in an appropriate context:

$$\frac{e \mapsto_a e'}{C[e] \mapsto_a C[e']}$$

(a) Context Lemma

Show that $\forall e,$ if e is a closed term, then $(e = v) \vee (e = D) \vee (\exists$ unique C, unique r_a, such that $C[r_a] = e)$, where we define $C[t]$ as

$$[\,][t] = t \qquad (C\ e)[t] = C[t]\ e \qquad (v\ C)[t] = v\ C[t] \qquad (\mathsf{try}\ C\ \mathsf{handle}\ e)[t] = \mathsf{try}\ C[t]\ \mathsf{handle}\ e$$

In English, show that for each closed term e in this language, it is either a value, of the form D, or that there is only one way to write down its evaluation context. This means that any expression e can be evaluated deterministically until we get a value v or a term of the form D (which must raise an exception).

(b) Equivalence of Evaluation Contexts

There is another way to define evaluation contexts for this language. We can remove D and add more possible redexes r_b as follows:

$$\begin{aligned} v &::= \ \lambda\ x\ e \\ C &::= \ [\,] \mid C\ e \mid v\ C \mid \mathsf{try}\ C\ \mathsf{handle}\ e \\ r_b &::= (\lambda\ x\ e)\ v \mid \mathsf{try\ raise\ handle}\ e \mid \mathsf{try}\ v\ \mathsf{handle}\ e \mid \mathsf{raise}\ e \mid v\ \mathsf{raise} \end{aligned}$$

The reductions for a redex r_b are these:

$$(\lambda\ x\ e)\ v \mapsto_b e\ \{v/x\}$$
$$\mathsf{try\ raise\ handle}\ e\ \mapsto_b e$$
$$\mathsf{try}\ v\ \mathsf{handle}\ e\ \mapsto_b v$$
$$\mathsf{raise}\ e \mapsto_b \mathsf{raise}$$
$$v\ \mathsf{raise} \mapsto_b \mathsf{raise}$$

Problem: show that all evaluations in the original semantics are also permitted in the more verbose semantics: $e \mapsto_a e' \Rightarrow e \mapsto_b^* e'$.

3. Iterators

Consider a binary tree defined by the Java class Tree:

```
class Tree {
    Tree left;
    Key here;
    Tree right;
}
```

We'd like to define an iterator that returns the elements in the tree in pre-order. The iterator might be used like this:

```
TreeIter i = new TreeIter(tree);
for (Key k = i.next(); k != null; k = i.next()) {
    // do something with k
}
```

It's actually quite awkward to implement TreeIter. We must explicitly save the state of the iteration in a stack between calls to `next()`. For instance,

```
class TreeIter {
  Stack stack;

  TreeIter(Tree t) {
    stack = new Stack();
    stack.push(t);
  }

  Key next() {
    while (! stack.empty()) {
      Tree t = (Tree) stack.pop();

      if (t.left == null && t.right == null) {
        return t.key;
      }

      if (t.right != null) stack.push(t.right);
      stack.push(new Tree(null, t.key, null));
      if (t.left != null) stack.push(t.left);
    }

    return null;
  }
}
```

Writing Java iterators for more complex data structures is even more difficult to get right. By contrast, the language CLU (c. 1977) has a powerful iterator construct that allows iterators to be expressed more elegantly. The iterator construct adds two expression forms: for x in e_i do e_b and yield e. The idea is that a for expression evaluates the "iterator" e_i. If a yield e_v expression is encountered during the evaluation of e_i, the body of e_b is evaluated in an environment where x is bound to the result of the evaluation of e_v. After evaluating e_b, control is transferred back into e_i at the point of the yield and evaluation continues as though the yield had resulted in e_b. A yield expression not in the scope of any for expression should raise a runtime error. Note that the variable x is bound in e_b but not in e_i.

If Java had CLU-style iterators, the binary tree iterator above could be written simply:

```
elements() yields Key {
    for k in left.elements() do { yield k };
    yield key;
    for k in right.elements() do { yield k }
}
```

And it could be used conveniently too:

```
for k in tree.elements() do {
    // do something with k
}
```

Consider a version of uF augmented with for and yield expressions:

$$\begin{aligned} e ::= {} & n \mid x \mid \#\mathsf{t} \mid \#\mathsf{f} \mid \#\mathsf{u} \\ & \mid \lambda x\ e \mid e_0\ e_1 \mid \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1 \\ & \mid \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\ & \mid \mathsf{for}\ x\ \mathsf{in}\ e_i\ \mathsf{do}\ e_b \mid \mathsf{yield}\ e \end{aligned}$$

(a) Extend the contextual operational semantics for uF to include for and yield expressions.

(b) Give a translation of this language into standard uF in continuation-passing style.

4. Continuity

Consider the function

$$minall : (\omega_\perp \to \omega_\perp) \to \omega_\perp = \lambda f : \omega_\perp \to \omega_\perp\ .\ \min_{y \in \omega} f(y)$$

In this definition, ω denotes the whole numbers (0, 1, 2, ...), and the function min gives the smallest number in all of the $f(y)$, or $\perp$ if $f(y) = \perp$ for some integer y. Show that this function is monotonic but not continuous.

5. Dynamic vs. Static Scope

Consider the following uF program. What value does it have under dynamic scope? Under the standard block-structured static scope? Explain briefly (don't show the whole evaluation).

```
let f = λnλg1λg2 (
  let x = n+10 in
   let g = (λz x) in
    if n==0 then (g #u) + (g1 #u) + (g2 #u)
             else (f (n-1) g g1)
) in
  (f 2 #u #u)
```

6. Implementation

Call-by-name evaluation often results in redundant computation. For instance,

$$\begin{aligned} & \mathsf{let}\ x = 3 + 2\ \mathsf{in} \\ & \quad \mathsf{let}\ y = x + 1\ \mathsf{in} \\ & \quad\quad y + y \end{aligned}$$

In this example, substituting 3 + 2 for x in the body of the let will cause 3 + 2 to be computed twice. With call-by-value semantics x and y are computed only once, but a computation may diverge (although not in this example) even if the result of the computation is not used. Since call-by-name is inefficient, most lazy functional languages such as Haskell or Miranda implement *call-by-need.* Call-by-need evaluation has the observational behavior of call-by-name (assuming expressions have no side effects) but requires no more substitution steps than call-by-value evaluation. Call-by-need is often implemented by overwriting an argument to an application with its value the first time it is evaluated, thus avoiding the need to subsequent re-evaluation. In the above program, the first evaluation of y will evaluate $x + 1$ and result in the value 6. Subsequent demands for the computation $x + 1$ will immediately return 6.

In this problem, you will write a translator from a call-by-need Scheme-like language into a call-by-value language.

The implementation files for this question are found in the file `need.tar.gz`, which is available from the course web page. The archive contains an implementation of a interpreter for the call-by-need source language (described below) as well as an interpreter for the call-by-value target language. You can run the interpreters and your translation using the `need` command.

The syntax for the full source language is given by:

$$
\begin{array}{rcl}
\mathit{op} & ::= & \texttt{+} \mid \texttt{*} \mid \texttt{and} \mid \texttt{or} \\
\mathit{binop} & ::= & \texttt{-} \mid \texttt{=} \mid \texttt{<} \\
\mathit{unop} & ::= & \texttt{zero?} \mid \texttt{left} \mid \texttt{right} \\
\mathit{refop} & ::= & \texttt{(ref } e\texttt{)} \mid \texttt{(! } e\texttt{)} \mid \texttt{(:= } e_1\ e_2\texttt{)} \mid \texttt{(seq } e_1\ e_2\ \ldots\ e_n\texttt{)} \\
e & ::= & x \mid n \mid \texttt{true} \mid \texttt{false} \mid \texttt{(}\mathit{op}\ e_1\ e_2\ \ldots\ e_n\texttt{)} \mid \texttt{(}\mathit{binop}\ e_1\ e_2\texttt{)} \mid \texttt{(}\mathit{unop}\ e\texttt{)} \mid \mathit{refop} \mid \texttt{(if } e_0\ e_1\ e_2\texttt{)} \\
 & & \mid \texttt{(fn (}x_1 \ldots x_n\texttt{) } e\texttt{)} \mid \texttt{(}e_1\ e_2\ \ldots e_n\texttt{)} \mid \texttt{(let ((}v_1\ e_1\texttt{)}\ldots\texttt{(}v_n\ e_n\texttt{)) } e\texttt{)} \mid \langle e_1, e_2 \rangle
\end{array}
$$

In this syntax definition, x stands for variable identifiers, n ranges over the integers and $A_1 \ldots A_n$ stands for one or more occurrences of syntactic objects of the form A. Lambda abstractions with any number of arguments $x_1, \ldots, x_n$ are written `(fn` $(x_1 \ldots x_n)$ e`)`. Both user-defined functions and primitive operations are applied to their arguments using prefix notation, just as in the simple lambda calculus. Thus `(+ 3 4)` evaluates to 7, and `((fn (x) (x)) 37)` evaluates to `37`

Arithmetic and boolean operators first evaluate their arguments and then perform the operation. Subtraction (`-`) and the relational operators (`=` and <) expect exactly two integer arguments, while `+`, `*`, `and`, and `or` expects at least two, but possibly more, arguments. The unary operators expect one argument.

The operator `zero?` expects one argument and returns `#t` if it is an integer expression that evaluates to 0 and `#f` if it is any other value.

The `let` expression binds variables to expressions e_i that may be used in the body e. The binding of v_i is available in e_j when $j > i$, as in the Scheme `let*` expression.

Pairs are enclosed in angle brackets (e.g., $\langle e_1, e_2 \rangle$). Pairs are lazy, so $\langle$(+ 3 4),5$\rangle$ does not immediately evaluate to $\langle$7,5$\rangle$, and $\langle\Omega,\Omega\rangle$ is not a divergent program. The `left` and `right` operators return the left and right components of a pair, respectively, forcing the evaluation of those components.

After compiling the interpreter using `make` or `make.bat` you should be able to load and run programs using the command `need`. For this assignment, you will need to modify only the file `translate.ml`. Indeed, this is the only file you should modify.

To simplify the AST somewhat, we desugar multi-argument functions into unary functions and we multi-argument primitive operations into binary operations, arriving at a core source language:

$$
\begin{array}{rcl}
\mathit{binop} & ::= & \texttt{+} \mid \texttt{*} \mid \texttt{-} \mid \texttt{=} \mid \texttt{<} \\
\mathit{unop} & ::= & \texttt{zero?} \mid \texttt{left} \mid \texttt{right} \\
\mathit{refop} & ::= & \texttt{(ref } e\texttt{)} \mid \texttt{(! } e\texttt{)} \mid \texttt{(:= } e_1\ e_2\texttt{)} \mid \texttt{(seq } e_1\ e_2\texttt{)} \\
e & ::= & x \mid n \mid \texttt{true} \mid \texttt{false} \mid \texttt{(}\mathit{binop}\ e_1\ e_2\texttt{)} \mid \texttt{(}\mathit{unop}\ e\texttt{)} \mid \mathit{refop} \mid \texttt{(if } e_0\ e_1\ e_2\texttt{)} \\
 & & \mid \texttt{(fn (}x\texttt{) } e\texttt{)} \mid \texttt{(}e_1\ e_2\texttt{)} \mid \texttt{(let ((}v_1\ e_1\texttt{)) } e\texttt{)} \mid \langle e_1, e_2 \rangle
\end{array}
$$

The core source language is just the full source language without multi-argument functions and applications, without most boolean operations, and without list operations. Note that `let` only defines one parameter.

Our target language has the same syntax except that *refop* expressions are not permitted. Expressions in the target language are also evaluated call-by-value rather than call-by-name.

You should write a translator from the core source language to the target language. A few cases for the translator we wrote are filled in in the source files for this problem, but you are free to do this translation any way you wish (as long as it requires modifying only `translate.ml`). To help you to determine whether your translation is correct, we have provided both a call-by-need interpreter and a call-by-value interpreter. Both interpreters are invoked by running `need` from the command line.

There are many possible ways to do the translation, but one way we suggest is to do it in two phases. In the first phase, translate the call-by-need language to call-by-value, but do not eliminate *refop* expressions. In the second stage, translate the call-by-value language with refs to a call-by-value language without refs. The first phase of the translation should be performed by the `Translate.to_call_by_value` function. The second phase of the translation should be performed by the `Translate.remove_refs` function. If you implement both phases of the translation at once, both phases should be performed in `Translate.to_call_by_value` and `remove_refs` should just be the identity function.

(a) Write a program in the source language that evaluates to a different result in call-by-value, call-by-need, and call-by-name. Submit your program by email with your version of `translate.ml`. You'll get extra credit if your program exposes bugs in other students' translations.

(b) Implement `Translate.to_call_by_value` in `translate.ml`. This function should translate a call-by-need input program into a call-by-value program that produces the same result. The most straightforward way to perform this translation is to use refs as described above, but you are not required to do so.

(c) Implement `Translate.remove_refs` in `translate.ml`. This function should remove all Ref, Deref, Assign, and Seq nodes from the AST. If your `to_call_by_value` translation already removed these nodes, `remove_refs` should be the identity function.