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.sml` to `jcheney@cs.cornell.edu` by 5PM on that day.

1. Continuity

   (a) 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.

   (b) Winskel, 8.16

2. Uniformity of fix

   The Fixed Point Theorem established that fix gives the *least* fixed-point of a continuous function on a pointed cpo. It is natural to ask whether or not other fixed-points could be used in defining denotational semantics. In one sense, we require the least defined function satisfying a given recursive equation; other fixed-points defined at more values may correspond to non-computable denotations. However, in another sense, taking the least fixed-point yields a canonical solution. A *fixed-point operator* F is a collection of continuous functions $F_D : [D \to D] \to D$ such that, for each cpo D and continuous function $f : D \to D$, $F_D(f) = f(F_D(f))$. In this problem, you will show that fix is the unique *uniform* fixed-point operator over pointed cpos.

   (a) Let D and E be pointed cpos. Suppose $f : D \to D$ and $g : E \to E$ are continuous functions and $h : D \to E$ is a strict continuous function (i.e., $h(\perp_D) = \perp_E$). Finally, suppose $h \circ f = g \circ h$. Show that for all $n \in \omega$, $h(f^n(x)) = g^n(h(x))$.

   (b) A fixed-point operator is *uniform* if for all continuous functions $f : D \to D$ and $g : E \to E$ and strict continuous function $h : D \to E$ such that $h \circ f = g \circ h$ we have $h(F_D(f)) = F_E(g)$. Prove that fix is uniform over pointed cpos.

   (c) Prove that fix is the unique uniform fixed-point operator over pointed cpos. (*Hint*: If D is a pointed cpo and $f : D \to D$ is a continuous function, then consider $D' = \{x \mid x \sqsubseteq \mathsf{fix}(f)\}$ and the restriction f' of f to D'.)

3. Parameter Passing and Scoping

   (a) Give a single REC$^+$ program that has a different meaning in each of call-by-name, call-by-value, and call-by-denotation. To check your answer, explicitly sketch the reduction of each denotation to normal form (you need not show every reduction along the way).

   (b) Lew Scannon has the idea that REC$^+$ programs can be protected from compilers and interpreters that use an parameter-passing style by having the program test itself to determine which style is being used, and then branching to an appropriate subprogram. Why won't Lew's idea work?

4. Implementation

   In this problem you will translate a lazy Scheme-like language with some interesting features into a simple variant of the untyped lambda calculus.

   The implementation files for this question are found in the file `lambda.tar.gz`, which is available from the course web page. The archive contains the implementation for an interpreter for the source language of your translation (which is described below). You can interact with the interpreter via four functions defined in `top.sml`. They are:

`load : string -> LambdaAst.expr` Takes the name of a file containing a lambda expression and returns the ML value for the program

`run : LambdaAst.expr -> LambdaAst.expr` Evaluates a lambda expression

`trans : LambdaAst.expr -> LambdaAst.expr` An interface to your translation described below

`run2 : LambdaAst.expr -> LambdaAst.expr` Runs translated expressions

The syntax for the full language is given by:

$$
\begin{array}{rcl}
op & ::= & \texttt{+} \mid \texttt{*} \mid \texttt{-} \mid \texttt{zero?} \mid \texttt{fst} \mid \texttt{snd} \\
 & & \mid \texttt{hd} \mid \texttt{tl} \mid \texttt{nil?} \mid \texttt{cons} \mid \texttt{\#t} \mid \texttt{\#f} \mid \texttt{if} \mid \texttt{and} \mid \texttt{or} \\
e & ::= & x \mid n \mid op \mid \texttt{(fn (}x_1 \ldots x_n\texttt{)}\ e\texttt{)} \mid \texttt{(}e_1 \ldots e_n\texttt{)} \mid \langle e_1, e_2 \rangle \mid \texttt{[}e_1 \ldots e_n\texttt{]} \\
 & & \mid \texttt{(left}\ e\texttt{)} \mid \texttt{(right}\ e\texttt{)} \mid \texttt{(case}\ e\ \texttt{(}v_1\ e_1\texttt{)}\ \texttt{(}v_2\ e_2\texttt{))} \\
 & & \mid \texttt{(letrec ((}v_1\ e_1\texttt{)} \ldots \texttt{(}v_n\ e_n\texttt{))}\ e\texttt{)}
\end{array}
$$

In this syntax definition, x stands for variable identifiers, n ranges over the integers and $A_1 \ldots A_n$ stands for zero 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 operators first evaluate their two arguments and then perform the operation. Subtraction ($-$) expects exactly two integers, while $+$ and $*$ work on any number of arguments. However, when $+$ and $*$ are used as expressions other than in the operator position in an application expression, they produce functions that expect exactly two arguments. Thus, the expression `(((fn (x) +) 0) 1 2 3)` does not have the same meaning as `(+ 1 2 3)`; it results in the application of a two argument function to three arguments (1, 2, and 3), which is an error.

Pairs are enclosed in angle brackets, so $\langle e_1\texttt{,}e_2 \rangle$ is a list containing three values. Pairs are lazy, so $\langle$`(+ 3 4),5`$\rangle$ does not immediately evaluate to $\langle$`7,5`$\rangle$. The `fst` and `snd` operators return the first and second components of a pair, respectively.

The `left` and `right` expressions, together with the `case` expression, are generalizations of boolean `true`/`false` and `if`. Both `left` and `right` expect one argument. The expressions `(left` e`)` and `(right` e`)` evaluate to themselves; the argument e is not evaluated at this point.

The `(case` e `(`v_1 e_1`) (`v_2 e_2`))` expression evaluates e. If it is of the form `(left` e'`)`, then e' is substituted for v_1 in e_1 and the result is evaluated. Similarly, if e evaluates to `(right` e'`)` then e' is substituted for v_2 in e_2. It is an error if e evaluates to anything else.

The boolean operators `if`, `and`, `or`, `#t`, `#f` are just "syntactic sugar" for special forms of `case` expressions. Each of these can be expressed as a combination of `left`, `right`, and `case` expressions. To see how, look at `ast.sml`. The list operations are also expressed in terms of pairs and cases, according to the (recursive) definition "list = either nil or cons of expression and list". These definitions are also in `ast.sml`.

You are free to use these syntactic sugar definitions in your translation as shortcuts, but keep in mind that they need to be translated too. For example, if you felt like representing some expression as `nil`, you could use the definition `LambdaAst.astNil`, but since this definition might contain additional constructs to be translated (such as `left`, `right`, and `case`), you would need to translate `astNil` also.

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 `letrec` expression binds variables to (potentially) mutually recursive expressions e_i that may be used in the body e. For example, the following function determines (slowly) whether an integer is even or odd.

```
(fn (z)
  (letrec ((even (fn (x) (if (zero? x) #t (odd (- x 1)))))
           (odd  (fn (x) (if (zero? x) #f (even (- x 1))))))
   (even z)))
```

After compiling the interpreter using `CM.make()` you should be able to load and run lambda programs using the commands described above. For this assignment, you will need to modify only the file `translate.sml`.

As described in class, it is possible to encode many concepts in the primitive lambda calculus that contains only variables, function application, and functions of a single variable. For this problem you will perform a translation of many primitives into the simple language. Since some of the constructs of the full language are just "abbreviations" for more complicated core expressions, you only need to translate the core expressions. You should be happy about this, since there are fewer cases for you to implement (and debug).

The core source language is:

$$
\begin{array}{rcl}
op & ::= & \texttt{+} \mid \texttt{*} \mid \texttt{-} \mid \texttt{zero?} \mid \texttt{fst} \mid \texttt{snd} \\
e & ::= & x \mid n \mid op \mid \texttt{(fn (}x_1 \ldots x_n\texttt{)}\ e\texttt{)} \mid \texttt{(}e_1 \ldots e_n\texttt{)} \mid \langle e_1, e_2 \rangle \\
 & & \mid \texttt{(left}\ e\texttt{)} \mid \texttt{(right}\ e\texttt{)} \mid \texttt{(case}\ e\ \texttt{(}v_1\ e_1\texttt{)}\ \texttt{(}v_2\ e_2\texttt{))} \\
 & & \mid \texttt{(letrec ((}v_1\ e_1\texttt{)} \ldots \texttt{(}v_n\ e_n\texttt{))}\ e\texttt{)}
\end{array}
$$

This is just the full language without list and boolean operations.

The syntax for the simple language is given by:

$$
\begin{array}{rcl}
op & ::== & + \mid - \mid * \\
e & ::== & x \mid n \mid \texttt{zero?} \mid \texttt{(fn (}x\texttt{)}\ e\texttt{)} \mid \texttt{(}e_1 e_2\texttt{)} \mid \texttt{(op}\ e_1\ e_2\texttt{)}
\end{array}
$$

where $+$ and $*$ now operate on exactly two integers, functions take exactly one argument, and there is no built-in support for pairs, cases, or recursive definitions. The binary operators $+$, $*$, and $-$ may not be used as expressions. Integers are supported as a built-in type in the simple language, although you could imagine translating them to Church numerals as shown in class.

You should write a translator from the core language to the simple 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.sml`). To help you to determine whether your translation is correct, there is a modified interpreter, which you can call by `run2`, to evaluate a lambda term in the simple language. The modified interpreter gives errors for programs that do not conform to the target language specification given above.

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, write a translator that handles everything but `letrec`, leaving `letrec` alone. You can test this translator using the original `run` interpreter. Then translate `letrec` expressions into other language constructs, and translate the result again. It's much easier (and less error-prone) to express `letrec` in terms of the other language constructs than to do the translation directly.