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 `zdance` by 5PM on that day.

1 Induction

Winskel, exercise 4.10. Do only the direction that was not shown in class: prove that any evaluations performed by the large-step semantics can be computed step-by-step in individual small-step transitions. Show only the cases of the `if` statement and the sequence of two commands, c_1 `;` c_2.

2 Free identifiers

a. Identify the free and bound variables in each of the following expressions:

   – $(\lambda \texttt{x}\ (\texttt{x}\ \texttt{y}))$
   – $(\lambda(\texttt{z}\ \texttt{x})\ (\texttt{x}\ \texttt{y}\ \texttt{z}))$
   – $(\texttt{z}\ (\lambda \texttt{y}\ ((\lambda \texttt{z}\ (\texttt{x}\ \texttt{y}))\ \texttt{z})))$

b. Suppose we add a `letrec` expression of the form described in problem 4 to the programming language. Write a rule for directly determining the free identifiers of this construct, using the free identifiers of its sub-expressions.

3 Encodings

a. Show how to write *AND*, *OR*, and *NOT* as lambda expressions in a manner consistent with the definitions of booleans given in class. Can you write lambda expressions for both short-circuit *AND* and the variety that evaluates both operands?

b. Show a sequence of β- and/or η-reductions that yields the normal form for the expression $(\mathit{INC}\ \texttt{2})$, using one of the definitions of *INC* given in class for Church numerals.

c. Show how to write *MULT*, *EXPT*, and *DEC* operations that work on Church numerals. Hint for *DEC*: first construct a function that maps a pair $\langle l, r\rangle$ to a pair $\langle l+1, l\rangle$.

4 Combinators

Consider the following definitions:

$$\begin{aligned} S &\equiv (\lambda(\texttt{x y z})\ ((\texttt{x z})\ (\texttt{y z}))) \\ K &\equiv (\lambda(\texttt{x y})\ \texttt{x}) \\ I &\equiv (\lambda \texttt{x}\ \texttt{x}) \end{aligned}$$

These are the S, K, and I *combinators*, which Curry experimented with to eliminate the need for variables. A combinator is just a lambda expression with no free variables. These three have the remarkable property that any lambda calculus expression containing no free identifiers (a *closed expression*) can be expressed using only applications operating on these three combinators, with no explicit variables or abstraction terms. This property also means that the lambda calculus can be universal with only three distinct identifier names, since no combinator uses more than three identifiers.

a. Reduce the following expressions to normal form:
   - `(K I)`
   - `(S K)`
   - `((K K) K)`
   - `(S (S S))`

b. Show that the I combinator is superfluous: the S and K combinators can be used to construct an expression with the same normal form.

c. Now, we will construct a translation from lambda expressions to expressions containing only applications of the S and K combinators. This translation will be defined in terms of two functions: $C[\![e]\!]$, which converts an expression e into this form, and a function $A[\![x,e]\!]$, which *abstracts* the variable x from the expression e. removing all uses of x within e.

   The idea is that $A[\![x,e]\!] = (\lambda\ x\ e)$, in the sense that the two expressions have the same effect when applied to any argument (they are extensionally equal). Using the function A, the function C can be defined simply.

$$\begin{array}{rcl} C[\![x]\!] & \equiv & x \\ C[\![e_0\ e_1]\!] & \equiv & (C[\![e_0]\!]\ C[\![e_1]\!]) \\ C[\![\lambda\ x\ e]\!] & \equiv & A[\![x, C[\![e]\!]]\!] \end{array}$$

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

$$A[\![x,x']\!] \equiv (K\ x') \qquad (x \neq x')$$

   Define the remainder of the translation to the S, K, and I combinators. Does this translation result in the most compact equivalent expression using the combinators?

5 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`, which takes the name of a file containing a lambda expression and returns the ML-value for program, `run : LambdaAst.expr -> LambdaAst.expr`, which evaluates a lambda expression, `trans : LambdaAst.expr -> LambdaAst.expr`, and interface to your translation described below, and `run2 : LambdaAst.expr -> LambdaAst.expr`, which runs translated expressions.

The syntax for the full language is given by:

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

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. The expressions `#t` and `#f` denote true and false, 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) #t)` evaluates to `#t`.

Arithmetic operators first evaluate their arguments and then perform the operation. Subtraction (`-`) expects exactly two integers, while `+` and `*` work on any number of arguments. The `if` operator expects exactly three arguments; after evaluating the the first to a boolean value, it conditionally evaluates its second or third argument. The `and` and `or` operators expect exactly two arguments; they evaluate their second argument conditionally upon the result of the first argument.

When the operators `+` 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.

Lists are enclosed in square brackets, so `[#t 1 (fn (x) x)]` is a list containing three values. Lists are lazy, so `[(+ 3 4)]` does not immediately evaluate to `[7]`. The `hd` and `tl` operators return the head and tail of a list, respectively, while `cons` prepends its first argument to its second (which must be a list).

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. Similarly `nil?` determines whether a list is empty, i.e. `[]`.

The `letrec` expression binds variables to (potentially) mutually recursive expressions e_i that may be used in the body e. The 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 constructs 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.

The syntax for the simple language is given by:

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

where now `+` and `*` operate on exactly two integers, functions take exactly one parameter, and there is no built-in support for lists, booleans, 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 full language to the simple language. To help 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.