Advance Programming Languages — Assignment 3
CS 6110 Spring 2010 — Due Monday, 15 March 2010

As before, you may work alone or in groups of two or three. If you work in a group, please form a group in CMS. Only one person needs to submit the assignment.

1. Well-Founded Relations

   Which of the following relations are well-founded? Justify your answers.

   (a) The relation $<$ on the integers $\mathbb{Z}$.

   (b) A relation $\prec$ on partial functions in $\mathbb{N} \rightharpoonup \mathbb{N}$, where

   $$f_1 \prec f_2 \quad \stackrel{\Delta}{\Longleftrightarrow} \quad \mathsf{dom}\, f_1 \subseteq \mathsf{dom}\, f_2 \ \wedge\ \forall x \in \mathsf{dom}\, f_1 \ f_1(x) < f_2(x).$$

   (c) Lexicographic (dictionary) order $<_{\text{lex}}$ on words over a finite ordered alphabet. Lexicographic order is defined as follows: given two words x and y over a finite ordered alphabet, $x <_{\text{lex}} y$ if either

   - x is a proper prefix of y, or
   - Neither x nor y is a prefix of the other, and at the first position at which they differ, the letter of x is less than the corresponding letter of y.

2. Static and Dynamic Scope

   What is the value of the following FL program under call by value with (a) static scoping, (b) dynamic scoping? Justify your answers.

   $\mathsf{let}\ x = 3\ \mathsf{in}$
   $\mathsf{let}\ f = \lambda y.\, x + y\ \mathsf{in}$
   $\mathsf{let}\ x = 4\ \mathsf{in}$
   $\mathsf{let}\ g = \lambda z.\, \mathsf{let}\ x = 5\ \mathsf{in}\ f\, x\ \mathsf{in}$
   $g\, x + f\, x$

3. Mutual Fixpoints

   The FL language has mutually recursive functions defined by

   $$\mathsf{letrec}\ f_1 = \lambda x.\, e_1\ \mathsf{and}\ \cdots\ \mathsf{and}\ f_n = \lambda x.\, e_n\ \mathsf{in}\ e.$$

   The bodies e_i may contain occurrences of any f_j, thus the functions may call one another and themselves recursively. We need to construct mutual fixpoints f_i such that $f_i = F_i\, f_1\, f_2 \ldots f_n$, where $F_i = \lambda f_1 \ldots f_n.\, \lambda x.\, e_i$, $1 \le i \le n$. To handle this in the λ-calculus, we need mutual fixpoint operators Y_i^n, $1 \le i \le n$, such that for any $F_1, \ldots, F_n$, the set $\mathsf{Y}_i^n\, F_1\, F_2 \ldots F_n$, $1 \le i \le n$ are mutual fixpoints of the F_i in this respect.

   (a) Give λ-terms Y_1^2 and Y_2^2 such that for any λ-terms F and G, $\mathsf{Y}_1^2\, F\, G$ and $\mathsf{Y}_2^2\, F\, G$ satisfy $\mathsf{Y}_1^2\, F\, G = F\, (\mathsf{Y}_1^2\, F\, G)\, (\mathsf{Y}_2^2\, F\, G)$ and $\mathsf{Y}_2^2\, F\, G = G\, (\mathsf{Y}_1^2\, F\, G)\, (\mathsf{Y}_2^2\, F\, G)$.

   (b) Generalize (a) to Y_i^n, $1 \le i \le n$. For full credit, strive for elegance and symmetry.

   Justify your answers.

4. Parameterized Fixpoints

   In the λ-calculus, one can apply the fixpoint combinator Y to a term with free variables, and this gives a definition of a fixpoint that works uniformly for all specializations of the free variables. Not in OCaml though: OCaml does not allow recursive function definitions $\mathsf{letrec}\ f = \lambda x.\, e$ (or any function definition for that matter) if e contains any free variables not in scope.

```
# let rec f x = if x = 0 then 0 else a + f (x - 1);;
Error: Unbound value a
```

However, we can parameterize the free variables and write $\mathsf{let}\ h = \lambda\overline{a}.\,\mathsf{letrec}\ f = \lambda x.\,e\ \mathsf{in}\ f$, where the free variables of $\lambda x.\,e$ are all among $\overline{a} = a_1, \ldots, a_n$ and f.

```
# let h a = let rec f x = if x = 0 then 0 else a + f (x - 1) in f;;
val h : int -> int -> int = <fun>
# h 611 10;;
- : int = 6110
```

Then we have

(*) For any set of values $\overline{v} = v_1, \ldots, v_n$, $(h\, v_1 \cdots v_n)$ evaluates to a fixpoint of $\lambda f x.\,(e\{\overline{v}/\overline{a}\})$.

But this formulation requires us to instantiate $\overline{a}$ before taking the fixpoint. We have no assurance that the calculation of the fixpoint does not depend on the choice of parameters $\overline{v}$ in some way. It would be nice if, like the λ-calculus, we could take the fixpoint first to get a solution that works uniformly for all instantiations of $\overline{a}$, then specialize by instantiating the $\overline{a}$. Show that this can always be done. Define $\mathsf{letrec}\ h = \lambda\overline{a}.\,\lambda x.\,(e\{(h\, a_1 \cdots a_n)/f\})$ and show that (*) holds for this definition of h.

```
# let rec h a x = if x = 0 then 0 else a + h a (x - 1);;
val h : int -> int -> int = <fun>
# h 611 10;;
- : int = 6110
```

5. λ-Lifting/Closure Conversion

In this programming exercise, you will demonstrate that nested λ-expressions are not essential to the expressiveness of a functional language. This is a useful result in compiling, because the language C, which is the target language of many compilers, does not allow nested functions.

We will define two versions of the FL language, a source version and a more restricted target version. The source language is defined by the following grammar:

$$\begin{array}{rcl} e & ::= & x \mid \lambda x_1, \ldots, x_n.\,e \mid e_1\, e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\ & & \mid \mathsf{letrec}\ f = e_1\ \mathsf{in}\ e_2 \mid \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid n \mid \mathsf{true} \mid \mathsf{false} \\ & & \mid e_1 + e_2 \mid e_1 \le e_2 \mid \ldots \text{ (other arithmetic and boolean expressions)} \end{array}$$

The target language is more primitive, allowing functions to be declared only at the top level of a program p, and only in a very restricted form.

$$\begin{array}{rcl} p & ::= & \mathsf{let}\ h = \lambda x_1, \ldots, x_n.\,e\ \mathsf{in}\ p \mid \mathsf{letrec}\ h = \lambda x_1, \ldots, x_n.\,e\ \mathsf{in}\ p \mid e \\ e & ::= & x \mid e_1\, e_2 \mid \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid n \mid \mathsf{true} \mid \mathsf{false} \\ & & \mid e_1 + e_2 \mid e_1 \le e_2 \mid \ldots \text{ (other arithmetic and boolean expressions)} \end{array}$$

Your job is to translate the source to the target.

Note that expressions e are λ-*free*—they may not contain λ-abstractions, let, or letrec. In effect, all λ-abstractions have been *lifted* to the outermost level. Moreover, these named λ-abstractions must be closed and their bodies λ-free.

Given an expression e, the translation is performed in two steps. The first step is called *closure conversion*. In this step, we convert each nested λ-abstraction, let expression, and letrec expression to a closed function F that takes the values of its free variables $a_1, \ldots, a_n$ as extra arguments. The function F will be bound to a fresh variable f at the top level. The occurrence of the original function in e is replaced by a function call $(f\, a_1 \cdots a_n)$. This computation is done from the inside out (i.e., smallest subterms first). The result is an environment σ consisting of a set of bindings and a λ-free expression e' of the target language.

The following are the formal rules that govern this translation for variables, application, and λ-abstraction with one variable (you will have to determine how to do the remaining constructs). The meta-expression $\mathcal{L}[\![e]\!]\sigma$ represents the result of translating e with prior bindings σ. The value of this expression is a pair $\langle e', \sigma' \rangle$, where e' is e with all nested functions extracted and σ' is the new environment with those functions bound to fresh variables and appended to σ.

$$\mathcal{L}[\![x]\!]\sigma = \langle x, \sigma \rangle \qquad \frac{\mathcal{L}[\![e_1]\!]\sigma = \langle e_1', \sigma' \rangle \qquad \mathcal{L}[\![e_2]\!]\sigma' = \langle e_2', \sigma'' \rangle}{\mathcal{L}[\![e_1\, e_2]\!]\sigma = \langle e_1'\, e_2', \sigma'' \rangle}$$

$$\frac{\mathcal{L}[\![e]\!]\sigma = \langle e', \sigma' \rangle,\ FV(\lambda x.\, e') = \{a_1, \ldots, a_n\}}{\mathcal{L}[\![\lambda x.\, e]\!]\sigma = \langle (f\, a_1 \cdots a_n),\ \sigma'[\lambda a_1, \ldots, a_n x.\, e'/f] \rangle} \text{ where } f \text{ is fresh.}$$

Most of the action is in the last rule. Given a λ-abstraction $\lambda x.\, e$, we recursively extract the functions in the body e, leaving the λ-free term e'. These functions are bound to fresh variables and the bindings are appended to σ to get σ'. That is the premise $\mathcal{L}[\![e]\!]\sigma = \langle e', \sigma' \rangle$. Now we form $\lambda x.\, e'$ and we wish to lift that as well. We find the free variables $a_1, \ldots, a_n$ and parameterize by them to form $\lambda a_1, \ldots, a_n x.\, e'$, which is now a closed term. This we bind to a fresh variable f and append to the environment to get $\sigma'[\lambda a_1, \ldots, a_n x.\, e'/f]$. In place of the original λ-expression we put $(f\, a_1 \cdots a_n)$.

The second step is to construct the program in the target language that initially creates the top-level bindings, then evaluates the λ-free term in the resulting environment.

For example, consider the following FL program with some nested function definitions.

```
let add3 =
  let compose f g x = f (g x) in
  compose (fun x -> x + 1) (fun x -> x + 2) in
add3 10
```

Here is a sample session with our solution code.

```
FL version 2010.0
>> load etc/test.txt
>> list
let add3 = let compose f g x = (f (g x)) in
((compose fun x -> (x + 1)) fun x -> (x + 2)) in
(add3 10)
>> run
13
>> lift
>> list
let a = fun add3 -> (add3 10) in
let b = fun x -> (x + 1) in
let c = fun x -> (x + 2) in
let d = fun c b compose -> ((compose b) c) in
let e = fun f g x -> (f (g x)) in
(a (((d c) b) e))
>> run
13
>> quit
bye
```

The archive `lifting.zip` contains an FL interpreter, some useful low-level infrastructure, and some sample FL programs. We have provided tools for creating fresh variables avoiding another set of variables, for finding all free variables and all variables in an expression, and safe substitution if you need it. There is also a representation of environments with lookup and update functions.

There are two functions `convert : exp -> state -> exp * state` and `lift : exp -> exp` in the file `lifting.ml` that you will have to implement. The function `convert` should implement the closure conversion step as described above, producing a λ-free expression e' and an environment σ. The function `lift` should call `convert` to get e' and σ, then produce the final output program in the target language from these. The resulting program should run under the FL interpreter provided.

The function `convert` consists mostly of easy one-line cases. The only difficult cases are `fun` and `letrec`. You can translate `let` to `fun`. The case `fun` is of moderate difficulty, but if you follow the rule above closely, you should have little trouble.

The case of `letrec` is quite difficult. Expect to spend most of your time on this case. Get everything else working first. Then for `letrec`, first get it working with top-level functions of the form $\lambda a_1, \ldots, a_n.\, \mathsf{letrec}\ f = \lambda x_1, \ldots, x_m.\, e\ \mathsf{in}\ f$. This is almost, but not quite, the required form. For the final ascent, use Exercise 4.

This is all you need to do for full credit on the assignment. But if you are up for a real challenge, for extra karma, implement a new interpreter command `convert` that does the closure conversion, but does not do the lifting. Instead, it evaluates the resulting λ-free expression in the resulting environment directly. For this, you will need to construct environments for the recursive functions that would have been defined with letrec, which contain circular references. You can use `State.rec_update` for this. Look at the implementation of letrec in the interpreter to see how `State.rec_update` is used.