Programming Languages (CS 329)
 Computer Science and Engineering
 Indian Institute of Technology Bombay

Midterm exam
 Books and personal notes permitted
 2001-09-21

This exam consists of pages 1 through 3. Credit for each question is marked up alongside and the total is given at the end. Use these credits for time management. It is advisable to provide some verbal intuition, because this may earn you partial credit in case the final answer is wrong. Some answers are posted in blue. Unless indicated there is no partial credit.

1. Similar to normal order and applicative order, we can impose many evaluation strategies on a lambda engine, even though not all of them may be useful for general evaluation of lambda expressions. Suppose we fire the following transformation specifically for Church numerals:

Find the leftmost outermost occurrence of $(E_1\ (E_1\ E_2))$ and change to $((\bar{2}\ E_1)\ E_2)$. (The check of equality of the two instances of E_1 is purely syntactic. It is not an equivalence test.)

   (a) Apply the above transformation to $\bar{7}$ as many times as possible and show the steps to the final result. $\boxed{1}$

$$\begin{aligned}
\bar{7} &= \mathtt{(}\lambda\ \mathtt{f\ (}\lambda\ \mathtt{a\ (f\ (f\ (f\ (f\ (f\ (f\ (f\ a)))))))))} \\
&= \mathtt{(}\lambda\ \mathtt{f\ (}\lambda\ \mathtt{a\ ((}\bar{2}\ \mathtt{f)\ (f\ (f\ (f\ (f\ (f\ a))))))\ ))} \\
&= \mathtt{(}\lambda\ \mathtt{f\ (}\lambda\ \mathtt{a\ ((}\bar{2}\ \mathtt{f)\ ((}\bar{2}\ \mathtt{f)\ (f\ (f\ (f\ a)))\ )\ )\ )\ )} \\
&= \mathtt{(}\lambda\ \mathtt{f\ (}\lambda\ \mathtt{a\ (\ (}\bar{2}\ \mathtt{(}\bar{2}\ \mathtt{f))\quad (f\ (f\ (f\ a)))\ )\ )\ )} \\
&= \mathtt{(}\lambda\ \mathtt{f\ (}\lambda\ \mathtt{a\ (\ (}\bar{2}\ \mathtt{(}\bar{2}\ \mathtt{f))\quad ((}\bar{2}\ \mathtt{f)\ (f\ a)\ )\ )\ )\ )}
\end{aligned}$$

   (b) Including the time to check syntactic equality, how much time (in big-O notation, such as $O(\log n)$ or $O(2^{\sqrt{n}})$) will be required for the transformation of $\bar{n}$ as a function of n? Give the tightest upper bound you can. Assume that one unit of time is needed to compare two nodes in an abstract syntax tree. $\boxed{1}$

   The total cost is $O(n \log n)$ assuming you can compare two subtrees for equality in constant time (this is possible by constructing some hash function while traversing the tree the first time). Thanks to Navneet Loiwal for correcting a mistake in an earlier answer.

   (c) Consider the transformed version, which we call $\bar{7}'$, which is of the form `(λ f (λ a ...))`. If its two arguments are `f` and `a`, then `a` appears exactly once as the operand of an application in the body of $\bar{7}'$ (as it does in $\bar{7}$). Write $\bar{8}$ by replacing `a` with `(f a)` in $\bar{7}'$. Now apply the transformation again and show the steps leading to a new expression $\bar{8}'$. $\boxed{1}$

$$\begin{aligned}
\bar{8} &= \mathtt{(}\lambda\ \mathtt{f\ (}\lambda\ \mathtt{a\ (\ (}\bar{2}\ \mathtt{(}\bar{2}\ \mathtt{f))\quad ((}\bar{2}\ \mathtt{f)\ (f\ \ }\underline{\mathtt{(f\ a)}}\ \mathtt{\ )\ )\ )\ )\ )} \\
&= \mathtt{(}\lambda\ \mathtt{f\ (}\lambda\ \mathtt{a\ (\ (}\bar{2}\ \mathtt{(}\bar{2}\ \mathtt{f))\quad ((}\bar{2}\ \mathtt{f)\ \ ((}\bar{2}\ \mathtt{f)\ a)\ )\ )\ )\ )} \\
&= \mathtt{(}\lambda\ \mathtt{f\ (}\lambda\ \mathtt{a\ (\ (}\bar{2}\ \mathtt{(}\bar{2}\ \mathtt{f))\ \ (\ (}\bar{2}\ \mathtt{(}\bar{2}\ \mathtt{f))\ \ a)\quad))\ )} \\
&= \mathtt{(}\lambda\ \mathtt{f\ (}\lambda\ \mathtt{a\ ((}\bar{2}\ \mathtt{(}\bar{2}\ \mathtt{(}\bar{2}\ \mathtt{f)))\ a)))}
\end{aligned}$$

   (d) In general, if `a` is replaced by `(f a)` in $\bar{n}'$ and the above transformation applied, how much time (in big-O notation) is needed to obtain $\overline{n+1}'$, including the time to check syntactic equality?

   All ASTs on the way have $O(\log n)$ nodes and there are at most $O(\log n)$ steps. So the total time is at most $O((\log n)^3)$. Full marks for this or better times (can improve to $O((\log n)^2)$), together with algorithm.

   (e) In terms of representing integers in the lambda engine, what have we achieved through this exercise? Be precise. $\boxed{1}$

   We learnt how to represent integers in logarithmic space, and how to implement `INC` in time that is polynomial in input size.

2. Evaluate the following FL! code using static scoping and

(a) Call-by-name $\boxed{1}$

(b) Call-by-value $\boxed{1}$

```
(let ( (a (new 0))
       (f (proc (x) (+ (read x) (read x)))) )
  (call f (begin (write a (+ (read a) 1)) a) ) )
```

3. Evaluate the following code using dynamically scoped FL. $\boxed{2}$

```
(let ( (x 1) )
  (let ( (f (proc () x))
         (g (proc (h) (call h))) )
    (let ( (x 2) )
      (call g f) ) ) )
```

4. Defining or understanding the meanings of FL! programs that use `rec` may be difficult because of side-effects.

   (a) Evaluate the following code using the SOS rule

$$\langle (\texttt{rec I } E), S\rangle \quad\Rightarrow\quad \langle [(\texttt{rec I } E)/\texttt{I}]E, S\rangle \tag{1}$$

   and normal order execution. $\boxed{1}$

```
(let ( (counter (new 0)) )
  (begin
    (call
      (rec fact
        (begin
          (write counter (+ (read counter) 1))
          (proc (n) (if (= n 0) 1 (* n (call fact (- n 1)))))
        )
      )    5    )
    (read counter) ) )
```

   (b) Redesign the `rec` rule (1) above so that the expression above evaluates to 1. $\boxed{2}$

$$\frac{\langle E, S\rangle \Rightarrow \langle E', S'\rangle}{\langle (\texttt{rec I } E), S\rangle \Rightarrow \langle (\texttt{rec I } E'), S'\rangle}$$

$$\langle (\texttt{rec I } V), S'\rangle \Rightarrow \langle [(\texttt{rec I } V)/\texttt{I}]V, S'\rangle.$$

5. The `with` construct for modules may be designed in (at least) two different syntactic forms. In one form, the bindings imported from a module are explicitly mentioned, in the form
`(with (`$\texttt{I}^+_{\text{import}}$`) from` E_{module} E_{body}`)`:

```
(let ( (pairs (module
                ( make-pair (proc (l r) ...) )
                ( get-left (proc (p) ...) )
                ( get-right (proc (p) ...) ) ) ) )
  (with (make-pair get-left) from pairs
    (let ( (ap (call make-pair 2 5)) )
      (call get-left ap) ) )  )
```

In the other form (discussed in class) *all* bindings available from a module are implicitly installed into the current environment:

```
(let ( (pairs (module
                ( make-pair (proc (l r) ...) )
                ( get-left (proc (p) ...) )
                ( get-right (proc (p) ...) ) ) ) )
  (with pairs
    (let ( (ap (call make-pair 2 5)) )
      (call get-left ap) ) )  )
```

The other constructs `module` and `select` remain as discussed in class.

(a) Desugar `module` and `select` into FL. $\boxed{2}$

(b) Give a simple desugaring of the first (explicit) form of `with` into FL. $\boxed{2}$

(c) Give a translation of the second (implicit) form of `with` into FL. (The translation is not as simple as desugaring, but this distinction is subjective.) $\boxed{2}$

6. What will the following C++ program print? $\boxed{3}$

```
class Base {
 public:
  int x;
  Base() : x(0) { }
  /*virtual*/
  void print() { cout << "Base " << x << endl; }
};
class Ext : public Base {
 public:
  Ext() { x = 1; }
  void print() { cout << "Ext " << x << endl; }
};
```

```
void main() {
  Base base;
  Base * pbase = &base;
  pbase->print();
  Ext ext;
  ext.print();
  pbase = (Base*) &ext;
  pbase->print();
}
```

Extra credit: What happens if we uncomment `/*virtual*/` above? $\boxed{1}$

$\boxed{\textbf{Total: 20}}$