**Programming Languages (CS 329)** | **Final examination**
**Computer Science and Engineering** | **2000-09-20**
**Indian Institute of Technology Bombay** | **(Open book, open notes)**

This exam consists of pages 1 through 4. 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.

1. The $\lambda$-grammar we have studied defines only single-argument abstractions. Under static scope, multi-argument abstractions can be curried into a cascade of single argument abstractions. E.g. `(lambda (f x) (f x))` can be curried into `(lambda (f) (lambda (x) (f x)))`.

   (a) Show with the smallest and simplest example you can find that the currying transformation is not valid with dynamic scope (2 points).

   (b) Design a simple mechanism using the lambda engine and some simple macros designed in class to support multi-argument abstractions with dynamic scope (2 points).

   Simply pass one parameter which is a list of all parameters, and modify `env-extend` to create bindings for many identifiers at the same time.

2. Write down a syntactic translation from pass-by-name FL to a pass-by-value FL interpreter. In particular give translations for `(lambda I` $E$`)`, `(`$E_1$ $E_2$`)` and `I` (4 points).

   As clarified during the exam, you are given a pass-by-value interpreter. You want to simulate a pass-by-name interpreter on top of it. The easiest way is to package arguments to functions within a `lambda` having a dummy parameter, and replace all mention of the argument with a call to the `lambda`. For example,

   ```
   ( (lambda (x) (if #t 5 x)) (/ 1 0) )
   ```

   would abort with an error, but

   ```
   ( (lambda (x) (if #t 5 (x #u))) (lambda (dummy) (/ 1 0) ) )
   ```

   will not.

3. What is the value of the following FL! expression? (4 points)

   ```
   (let r = (new 1)
     (let top = (label x x)
       (if (> (read r) 10)
         (read r)
         (begin
           (r := (* 2 (read r)))
           (goto top #u) )  )  )  )
   ```

   The answer, 16, is easy to guess, but not easy to explain. Ideally we should have allotted 2 points for the correct answer and 2 more for a clean explanation. Not to go back on our word of "full score for correct answer" we will award one extra point for a good explanation.

4. We wish to enhance the imperative flavour of FL! by adding a looping construct:

   $$E \;\; ::= \;\; \ldots \mid \texttt{(loop } E\texttt{)} \mid \texttt{(break } E\texttt{)}$$

   `(loop` $E$`)` evaluates $E$ repeatedly forever. `(break` $E$`)` ends the nearest lexically enclosing `loop` with the value of $E$. It is an error to evaluate a `break` outside a lexically enclosing `loop`.

   (a) Write an iterative factorial function using these constructs (2 points).

   (b) Give a syntactic translation of these constructs using the `callcc` construct as in Scheme (4 points).

   We will translate `loop` and `break` into `label` and `jump`, which we know can be written using `callcc`.

5. What is the value of the following FLOP expression under static scoping and left-to-right multiple inheritance? (6 points)

```
(let ob2 = (object (method value (self) 2))
     ob3 = (object (method value (self) 3))
  (let ob5 = (object
               (method value (self) 5)
               (method compute (self) (* (ob3.value) (value))) )
    ((object ob2 ob3 ob5).compute)   )   )
```

The answer is 6. Even though `ob5` responds to the `compute` message and `ob3.value` evaluates to 3, the underlined method call `value` is shorthand for `self.value`, where `self` is the current object, which is not `ob5`, but the composition of `ob2`, `ob3` and `ob5`. Therefore `ob2` responds with value 2, and the answer is $3 \times 2 = 6$. Note that lexical scope has nothing to do with this, because `self` is a parameter passed at call time.

You will get this right only if you handled the `self` variable in the correct way in the midterm question; this issue was not discussed later in class. Hence the other obvious answer, $3 \times 5 = 15$ (too easy!) will be given 3 marks.

6. In this question we will design an alternative syntax and denotational mechanism for exception handling. The idea is to let the programmer bind exceptions to identifiers explicitly.

Three new productions are added to the grammar:

- `(exception` $E$`)` creates an exception with value $E$. Any attempt to access a variable bound to an exception results in the immediate raising of the exception.
- `(letx I=`$E_1$ $E_2$`)` binds to `I` an exception or value as a result of evaluating $E_1$ and then evaluates $E_2$ with this binding.
- `(catch` $E_1$ $E_2$`)` evaluates to the value $E_2$ if it does not throw any exception. Otherwise the value of the expression is the result of evaluating $E_1$ (which is expected to be a one-parameter function) with the exception raised from $E_2$ as its argument.

For example, the code

```
(catch (lambda (x) (+ 1 x))
  (letx a=(exception 4)
    (let b=2
      (+ a b) ) ) )
```

evaluates to 5 because accessing `a` in the last line raises the exception bound to `a`. Complete the following standard denotational specification (12 points):

$$
\begin{array}{rcl}
\mathcal{E} & : & \text{Expr} \to \mathit{Env} \to \mathit{Computation} \\
\mathit{Computation} & = & \mathit{XCont} \to \mathit{ExpCont} \to \mathit{CmdCont} \\
w \in \mathit{XCont} & = & \mathit{ExpCont} \\
\delta \in \mathit{DenotVal} & = & \mathit{Value} + \mathit{Exception} \\
\mathit{Exception} & = & \mathit{Value} \\
\mathit{Proc} & = & \mathit{Value} \to \mathit{Computation} \\
\mathcal{E}[\![\texttt{(lambda I } E\texttt{)}]\!] & = & \lambda u\, \lambda w\, \lambda k \_\_\_\_ \\
\mathcal{E}[\![\texttt{(call } E_1\ E_2\texttt{)}]\!] & = & \_\_\_\_ \\
\mathcal{E}[\![\texttt{(exception } E\texttt{)}]\!] & = & \_\_\_\_ \\
\mathcal{E}[\![\texttt{(letx I=}E_1\ E_2\texttt{)}]\!] & = & \_\_\_\_ \\
\mathcal{E}[\![\texttt{(catch } E_1\ E_2\texttt{)}]\!] & = & \lambda u\, \lambda w\, \lambda k \left(\mathcal{E}[\![E_1]\!]\, u\, w\, (\lambda p \_\_\_\_)\right) \\
\mathcal{E}[\![\texttt{I}]\!] & = & \_\_\_\_
\end{array}
$$

7. Reconstruct the type of the following expression (6 points):

```
(letrec accumulate = (lambda (combiner seed lst)
                       (if (null? lst)
                         seed
                         (combiner (car lst)
                                   (accumulate combiner seed (cdr lst)) ) ) )
  accumulate )
```

The type (schema) of `accumulate` is

$$\forall s, t \quad (s \to t \to t) \to t \to (\texttt{listof}\ s) \to t \tag{1}$$

8. Here is an incomplete implementation of Quicksort in a declarative language such as Prolog without `cut` as discussed in class (`<=` and `>` are numeric primitive relations provided by the interpreter):

```
(qsort null null)
(qsort (cons X Y) Z) :-
    (and (partition X ____ Y1 Y2)
         (qsort  Y1 ____)
         (qsort  Y2 Z2)
         (append Z1 ____ Z) )
(partition X null null null)
(partition X (cons A B) (cons A Y1) Y2) :-
    (and (<= A X)
         (partition X B Y1 Y2) )
(partition X (cons A B) Y1 ____) :-
    (and (> A X) ____ )
```

(a) Complete the implementation (4 points). The sorted numbers should be in increasing order.

```
(qsort null null)
(qsort (cons X Y) Z) :-
    (and (partition X Y Y1 Y2)
         (qsort  Y1 Z1)
         (qsort  Y2 Z2)
         (append Z1 (cons X Z2) Z) )
(partition X null null null)
(partition X (cons A B) (cons A Y1) Y2) :-
    (and (<= A X)
         (partition X B Y1 Y2) )
(partition X (cons A B) Y1 (cons A Y2)) :-
    (and (> A X) (partition X B Y1 Y2) )
```

The apparently similar attempt

```
(qsort (cons X Y) Z) :-
    (and (partition X (cons X Y) Y1 Y2)
         (qsort  Y1 Z1)
         (qsort  Y2 Z2)
         (append Z1 Z2 Z) )
```

does not work; can you find out what goes wrong?

(b) Trace the steps taken the declarative execution engine initialized with the above rule base and given the following target to satisfy:

```
(qsort P (cons 1 (cons 2 (cons 3 null))) )
```

(4 points)

This was a wicked question. Based on our discussion of list operations in class, and the mention of 'reversibility' between input and output, you may be tempted to jump to the conclusion that `P` will be successively bound to all permutations of the elements of the list $(1, 2, 3)$. This won't happen with the rulebase as given. The only rule that will match the initial goal has consequent

```
(qsort (cons X Y) (cons 1 (cons 2 (cons 3 null))))
```

Our intermediate goals will therefore be, exactly in the order below:

```
(partition X Y Y1 Y2)
(qsort Y1 Z1)
(qsort Y2 Z2)
(append Z1 (cons 1 Z2) (cons 1 (cons 2 (cons 3 null))))
```

Note that the constraints with respect to provided constants (`Z`) are mentioned in the last goal, *before* the first rule involving `partition`. (Recall that in the search algorithm we have discussed, order is important.)

When the system seeks to satisfy the first goal `partition`, the primitive operation `<` will fail because the arguments are not instantiated to numbers. Thus the whole computation will fail. Try it out with the following Prolog rulebase:

```
qsort([], []).
qsort([X|Y], Z) :-
        partition(X, Y, Y1, Y2),
        qsort(Y1, Z1),
        qsort(Y2, Z2),
        append(Z1, [X|Z2], Z).
partition(_, [], [], []).
partition(X, [A|B], [A|Y1], Y2) :-
        A < X,
        partition(X, B, Y1, Y2).
partition(X, [A|B], Y1, [A|Y2]) :-
        A >= X,
        partition(X, B, Y1, Y2).
append([], W, W).
append([A|B], W, [A|V]) :-
        append(B, W, V).
```

and goal `qsort(U, [1,2,3]).`.

On the other hand, if you rearrange the antecedents of the second rule to read as:

```
qsort([X|Y], Z) :-
        append(Z1, [X|Z2], Z),
        qsort(Y1, Z1),
        qsort(Y2, Z2),
        partition(X, Y, Y1, Y2).
```

the behavior of the Prolog engine will be different! Try it out and explain it.

**Total points: 50**