CSC173 Module 3
Mid-Term Exam
With Answers and FFQ

Chris Brown

November 27, 2007

Please write your name on the bluebook. This is a closed-book Exam. There are 75 possible points (one per minute). Stay cool and please write neatly.

1 λ Expression Evaluation (25 mins)

1. Evaluate the following λ expression (Hint: six new lines).

   `(((λx.λy.λz.((x y) z) λf.λa.(f a)) λi.i) λj.j)`

   Answer:

```
(((λx.λy.λz.((x y) z) λf.λa.(f a)) λi.i) λj.j) =>
((λy.λz.((λf.λa.(f a) y) z) λi.i) λj.j) =>
(λz.((λf.λa.(f a) λi.i) z) λj.j) =>
((λf.λa.(f a) λi.i) λj.j) =>
(λa.(λi.i a) λj.j) =>
(λi.i λj.j) =>
λj.j =>
```

2. Show that the function (a) below is equivalent to the function resulting from expression (b) below by applying each the arbitrary argument `<arg>`. Hint: nothing but => ... => and one `=>` needed. Preserve functions as just their names (in fact, `sa` and `ss` are OK and quicker to write.) as long as possible, do minimal rewriting down to the λ level. (Hint: (a) is a one-liner, (b) is (for me) 7 lines with two λs appearing.)

   (a) `identity`
   (b) `(self-apply (self-apply select-second))`

Answer:

(a) `(identity <arg>) => ... =>`
`<arg>`

(b) ((self_apply (self_apply select_second)) <arg>) => ... =>
(((self_apply select_second) (self_apply select_second)) <arg>) =>... =>
(((select_second select_second) (self_apply select_second)) <arg>) => ... =>
((λsecond.second (self_apply select_second)) <arg>) => ... =>
((select_second select_second) <arg>) => ... =>
(λsecond.second <arg>) => ... =>
<arg>

2 λ Calculus Arithmetic and Recursion (15 mins)

Write a function `fib` that computes the nth Fibonacci number, defined as $fib(n) = fib(n-1) + fib(n-2); fib(0) = 1, fib(1) = 1$. In your definition you may use infix notation and can assume the following functions are defined: `one, succ, pred, iszero, equal, recursive, or, and, if...then...else, add, sub, mult, exp, true, false, greater`.

Here's a start: you fill in the .... (Hints: You don't need all those functions, and my answer is three lines).

```
def fib1 f n = ...
def fib = recursive fib1
```

Answer:

```
def fib1 f n =
 if ((iszero n) or (equal n 1))
  then one
  else (add (f (pred n)) (f (pred(pred(n)))))

def fib = recursive fib1
```

3 (Fairly) Short Answers (20 mins)

1. What is the Church-Turing Thesis?

   Answer:

   Church's Thesis: "Every effectively calculable function (effectively decidable predicate) is general recursive" (Kleene 1952:300)

   Turing's Thesis: "Turing's thesis that every function which would naturally be regarded as computable is computable under his definition, i.e. by one of his machines, is equivalent to Church's thesis by Theorem XXX." (Kleene 1952:376)

   Or: Church-Turing Thesis: All formal models of algorithmic computation invented since before computers existed turn out to be equally powerful, accepting and generating equivalent languages (Brown 2007).

2. What are normal order and applicative order evaluation?

   Answer:

   Applicative Order: In function application, all occurrences of a function's bound variable are replaced by the value of the argument expression. Like "call by value".

   Normal Order: In function application, all occurrences of a function's bound variable are replaced by the unevaluated argument expression. Like "call by name".

3. What do the two Church-Rosser Theorems say?

   Answer:

   Order is irrelevant to the question, but for the record:

   First Church-Rosser Theorem: If an expression is reduced using two different evaluation orders and both reductions terminate, then they produce the same final result. So we can pretty much use either evaluation order.

   BUT, there's the Second Church-Rosser Theorem: If any evaluation order will terminate, then normal order reduction is guaranteed to terminate.

   So normal order evaluation is better choice if we are concerned with termination (not necessarily efficiency).

4. How is the paradoxical combinator defined and what is it for?

   Answer:

   The paradoxical combinator is for implementing recursion in λ calculus. Because it is self-replicating it provides the copying mechanism whereby a copy of the function may be passed down to lower levels of recursion. The copy of its argument function that it emits when called is the one that is applied to "itself" to make the recursion work.

   $Y(f)$, or the paradoxical combinator, is defined by $Y(f) = fY(f)$. This should get about 75% credit.

   In more detail for full credit: in λ calculus, we defined the equivalent function `recursive` as:

   λf.(λs.(f (s s)) λs.(f (s s))),

   or in its raw form,

   ```
   def recursive f = λs.(f (s s)) λs.(f (s s))
   ```

5. How can we implement types and type-checking in λ calculus?

   Answer: The way we saw in class was to use integers to encode types (zero for error, one for boolean, ...), and to represent a typed object as a pair with the first member the type code and the second the (untyped) value. Then to get some form of dynamic type checking: to operate on two typed operands, we check their type codes to see if they agree with what the operation expects, then perform the untyped operation on the two values, then create and return a new typed object (pair) that has the appropriate type code and the value from the untyped operation.

4 Scheme Binding and Continuation (15 mins)

1. What does this Scheme expression evaluate to?

```
(let ([f (let ([x 4])
           (lambda (y) (+ x y)))])
    (let ([x 40])
      (f 2)))
```

   Answer: This question gets at the lexical-binding aspect of Scheme, which is different from LISP and is one of its defining characteristics. The third `let` does not affect the binding from the second `let`, which affects the `x` in the definition of `f` (accomplished by the first `let`). Thus the expression evaluates to 6 (six).

2. Given:

```
(define setstart #f)

(define flexfact
  (lambda (x)
   (if (= x 0)
    (call/cc (lambda (cont) (set! setstart cont) 1))
    (* x (flexfact (- x 1))))))
```

   a) What does `(flexfact 4)` evaluate to?

   b) Having done that, what does `(setstart 2)` evaluate to?

   Answer:

   Continuations are another key part of the Scheme paradigm.

   a) `(flexfact 4)` is 4! or 24.

   b) `(setstart 2)` uses continuation to set the base case of `flexfact` to 2 instead of 1, so we get 4*3*2*2 or 48.