CIS 624 Programming Languages

Spring 1999

Midterm (100 pts)

Name (please print): ....

Id: ....

1	Substitution	20 pts	
2	Environments as Functions	20 pts	
3	Recursion to Iteration	20 pts	
4	Y Combinator	20 pts	
5	Program Execution	20 pts	
X	Extra Credit	10 pts	
	Total	100 pts	

1 Substitution

The semantics of a language based on the λ-calculus can be expressed using substitution. To evaluate of a program `M`, one calls the function `eval` below which calls `substitute` to perform function applications:

```
eval M =
  case M of
    number => return number
    variable => ERROR
    (lambda x.N) => return (lambda x.N)
    (app M1 M2) => let v1 = eval(M1)
                       v2 = eval(M2)
                   in case v1 of
                        (lambda x.M3) => return eval (substitute x V2 M3)


substitute x V M =
  case M of
    number => return number
    variable => if (variable == x) then return V else return variable
    (app M1 M2) => return (app (substitute x V M1) (substitute x V M2))
    (lambda z.B) => let newVar = make_new_var_somehow
                        newBody = substitute z newVar B
                    in return (lambda newVar. substitute x V newBody)
```

For example, the evaluation of the program:

```
((lambda (x) (lambda (y) (x y))) (lambda (x) x))
```

proceeds as follows:

```
  eval ((lambda (x) (lambda (y) (x y))) (lambda (x) x))
= eval (substitute x (lambda (x) x) (lambda (y) (x y)))
= let newBody = substitute y z (x y) // pick newVar = z
  in eval (lambda z. substitute x (lambda (x) x) newBody)
= let newBody = (x z)
  in eval (lambda z. substitute x (lambda (x) x) newBody)
= eval (lambda z. substitute x (lambda (x) x) (x z))
= eval (lambda z. ((lambda (x) x) z))
```

- As apparent in the above specification, one constantly create new variables during evaluation, *i.e.*, at *at run time*. An obvious question would be whether it is possible to avoid this overhead by preprocessing our programs such that all variables are distinct before calling `eval`. If you believe it is possible outline how it could be done and give an informal argument why it would work. If you believe it is impossible, give an expression in which all variables are distinct that still requires the creation of new variables at run time.

- Even though this idea of using substitution to perform function calls seems like a theoretical waste of time, it is actually implemented in most optimizing compilers. Indeed when compilers spot a call to a known function with a simple argument, they often *inline* the function call. Inlining is just another name for performing β-reduction. Given your answer to the previous question, can a compiler get away with inlining functions without creating new names?

2 Environments as Functions

Traditional environments have two constructors `make_empty` and `make_extended` and one operation `lookup` such that the following axioms hold:

```
(lookup (make_empty) x) = ERROR
(lookup (make_extended env x v) x) = v
(lookup (make_extended env x v) y) = (lookup env y)
```

Here is a proposed implementation of environments in Scheme. Given that the primitive `eqv?` checks if two Scheme objects are identical, verify that this implementation is correct:

```
(define make_empty
  (lambda ()
    (lambda (x)
      (error 'make_empty "Unbound variable ~s~n" x))))

(define make_extended
  (lambda (env x v)
    (lambda (y)
      (if (eqv? x y)
          v
          (lookup env y)))))

(define lookup
  (lambda (env x)
    (env x)))
```

3 Recursion to Iteration

Here is a recursive definition of a function that multiplies two numbers:

```
(define mult
  (lambda (m n)
    (if (zero? m)
        0
        (+ n (mult (sub1 m) n)))))
```

- Convert the function `mult` to continuation-passing style (CPS) using functions to represent continuations. The initial continuation is represented by `(lambda (x) x)`.

- Rewrite the CPS version of `mult` to abstract the representation of continuations. In other words, replace every application of a continuation `k` to an argument `a` by `(apply k a)` where the function `apply` is defined as follows:

  ```
  (define apply (lambda (k a) (k a)))
  ```

  Also replace every continuation function, with a call to a constructor `makeK1`, `makeK2`, etc with the appropriate arguments. For example, the initial continuation is now represented by a call to `makeK1` with no arguments. The constructor `makeK1` returns the functional representation of the continuation:

  ```
  (define makeK1 (lambda () (lambda (x) x)))
  ```

- Establish the axioms that any representation of continuations must satisfy. One of the axioms is:

```
(apply (makeK1) v) = v
```

- Using the familiar axioms for arithmetic, *prove* that the CPS version of `mult` is equivalent to the following iterative implementation:

```
(define multiter
  (lambda (m n sum)
    (if (zero? m)
        sum
        (multiter (sub1 m) n (+ sum n)))))
```

4 Y Combinator

Here is the definition of the Y combinator of the λ-calculus:

$$Y = \lambda f.(\lambda x.f(x\ x))\ (\lambda x.f(x\ x))$$

- Let M be a λ-calculus term. *Prove* using the axioms of the λ-calculus that:

$$(Y\ M) = M\ (Y\ M)$$

- What would be the result of evaluating the following expressions:
  - $Y\ (\lambda x.x)$:
  - $Y\ (\lambda x.y)$:

5 Program Execution

Write the result of evaluating each of the following Scheme expressions:

- ```
  (let ((x 1))
    (let ((x 2)
          (y x))
      (+ x y)))
  ```
- ```
  (add1 (call/cc (lambda (k) (sub1 (k 3)))))
  ```
- ```
  (let ((x 1))
    (let ((f (lambda (y) (+ x y))))
      (let ((x 2))
        (f 4))))
  ```
- ```
  (let ((x 1))
    (let ((f (lambda (y) (let ((x y)) (lambda (g) (g (g x)))))))
      ((f 3) (let ((x x)) (lambda (z) (+ x z))))))
  ```
- ```
  (let ((x 1))
    (let ((f (lambda (y) (+ x y))))
      (f ((lambda (d) x) (set! x 6)))))
  ```
- ```
  (letrec ((fact (lambda (n r)
                   (if (zero? n)
                       r
                       (fact (sub1 n) (* n r)))))
           (memoized_fact (let ((arg 0) (result 1))
                            (lambda (n)
                              (if (= arg n)
                                  result
                                  (let ((r (begin
                                             (printf "Calling fact~n")
                                             (fact n 1))))
                                    (begin (set! arg n)
                                           (set! result r)
                                           result)))))))
    (begin
      (memoized_fact 5)
      (memoized_fact 5)
      (memoized_fact 5)))
  ```

Extra Credit

An object (in the OOP sense) is an entity that encapsulates some information (instance variables) and associated operations (methods). For example, a stack object could encapsulate a list of positive numbers such that the only operations that could be performed on this list are:

- `empty?`: which checks if the list is empty.
- `push(n)`: which adds the number `n` to the beginning of the list.
- `top`: which returns the top element of the list (if the list is empty we will just return -1).
- `pop`: which deletes the first element of the list if there is one, and otherwise has no effect.

Using the ideas discussed in class about how to encode data structures in the λ-calculus, encode a stack object as a Scheme function. Is there any advantage to the Java representation of the stack object versus your Scheme representation?