Mid-Term Exam

May 9, 1996

General Information

The exam consists of 8 problems worth 100 points and an extra credit problem worth 5 points.

Rules

- You have three *consecutive* hours to complete the exam.
- You can consult Friedman, Wand, and Haynes and your lecture notes. Do not consult any other books. Do not talk to anybody else about the exam even after you turn in your solution.
- Write your answers in the space provided. Attach additional pages if you need to.
- The exam is due back in my office before noon on May 14.

Some of the problems refer to incomplete interpreters. In particular I have omitted the definitions for the initial environment, `lookup`, `extend`, and similar "obvious" help functions. Do not bother defining these procedures.

Before you start ...

Fill in your name and start time *now*. Do not forget to write down your finish time when you are done.

- Name:
- Start Time:
- Finish Time:

Good luck!

Problem 1 (12 points)

Consider the function `prodprimes` that computes the product of all primes less than or equal to its argument:

```
(define prodprimes
  (lambda (n)
    (if (= n 1)
        1
        (if (prime? n)
            (* n (prodprimes (sub1 n)))
            (prodprimes (sub1 n))))))
```

- (6 points) Convert `prodprimes` to CPS assuming that the function `prime?` is simple, i.e., the call to `prime?` should be left intact.

- (6 points) Convert `prodprimes` to CPS assuming that the function `prime?` has also been converted to CPS.

Problem 2 (10 points)

The abstract syntax for `while` loops is:

```
(define-record while (condition body))
```

The semantics of `while` loops is completely specified using the following Scheme macro:

```
while C do B =>
(letrec ([loop (lambda ()
                 (if C
                     (begin B (loop))
                     0))])
  (loop))
```

Your task is to complete the `while` clause in the following interpreter. Do *not* add any other clauses to the interpreter!

```
(define eval
  (lambda (exp env)
    (variant-case exp
      [lit (datum) datum]
      [varref (var) (unbox (lookup env var))]
      [if (test then else)
        (if (true? (eval test env)) (eval then env) (eval else env))]
      [varassign (var exp) (set-box! (lookup env var) (eval exp env))]
      [begin (exp1 exp2) (begin (eval exp1 env) (eval exp2 env))]
      [proc (formal body) (lambda (arg) (eval body (extend env formal arg)))]
      [app (rator rand) ((eval rator env) (eval rand env))]
      [while (condition body)









      ]
      )))
```

Problem 3 (6 points)

Convert the following function to CPS:

```
(define A
  (lambda (m n)
    (if (zero? m)
        (add1 n)
        (if (zero? n)
            (A (sub1 m) 1)
            (A (sub1 m) (A m (sub1 n)))))))
```

Problem 4 (15 points)

Here is an interpreter for a dynamically-scoped variant of our language:

```
(define eval
  (lambda (exp env)
    (variant-case exp
      [lit (datum) datum]
      [varref (var) (lookup env var)]
      [proc (formal body) (make-dynproc formal body)]
      [if (test then else)
        (if (zero? (eval test env))
            (eval then env)
            (eval else env))]
      [let (var rhs body) (eval body (extend env var (eval rhs env)))]
      [app (rator rand) (apply-proc (eval rator env) (eval rand env) env)])))

(define apply-proc
  (lambda (fun arg env)
    (variant-case fun
      [dynproc (formal body) (eval body (extend env formal arg))]
      [primitive (p) (p arg)])))
```

- (7 points) Using the interpreter, trace the execution of the following program until the *first* call to `apply-proc`. You are only required to show the values of `fun`, `arg`, and `env` that are passed to that first call to `apply-proc`.

```
let countdown = proc (n)
                  if n
                  then 0
                  else countdown(sub1(n))
in countdown(2)
```

- (8 points) Make sure you understand the answer to the previous question. Now extend the interpreter with a clause for user-defined recursive procedures. The abstract syntax is:

  ```
  (define-record recproc (formal body))
  ```

Problem 5 (12 points)

Consider the following two definitions:

```
(define f1
  (lambda (ls)
    (if (null? ls)
        0
        (SOME-OPERATION (car ls) (f1 (cdr ls))))))

(define f2
  (lambda (ls)
    (letrec ([help (lambda (ls result)
                     (if (null? l)
                         result
                         (help (cdr l) (SOME-OPERATION result (car l)))))])
      (help ls 0))))
```

Give sufficient conditions on `SOME-OPERATION` to guarantee that the two definitions are equivalent. Which of the following operations satisfy your conditions: addition, substraction, multiplication, and division?

Problem 6 (20 points)

A stream is a potentially infinite list, *e.g.*, the standard input and standard output files. The abstract syntax for stream operations is:

```
(define-record scons (head tail))
(define-record scar (stream))
(define-record scdr (stream))
```

The semantics of these new expressions is completely specified using the following Scheme macros:

```
(scons M N) => (cons M (lambda (_) N))
(scar M) => (car M)
(scdr M) => ((cdr M) 42)
```

The identifier `_` is a dummy formal parameter that is not used in `N`.

- (5 points) *prove* using the axioms for `cons`, `car`, and `cdr` that:

  ```
  (scar (scons M N)) = M
  (scdr (scons M N)) = N
  ```

- (15 points) complete the missing three clauses of the following interpreter. Again do not add any other clauses to the interpreter.

```
(define apply-proc
  (lambda (fun arg)
    (variant-case fun
      [closure (formal body env) (eval body (extend env formal arg))]
      [primitive (p) (p arg)])))

(define eval
  (lambda (exp env)
    (variant-case exp
      [varref (var) (lookup env var)]
      [proc (formal body) (make-closure formal body env)]
      [app (rator rand) (apply-proc (eval rator env) (eval rand env))]
      [scons (head tail)







      ]
      [scar (stream)







      ]
      [scdr (stream)







      ]
```

Problem 7 (15 points)

In CPS, every "interesting" procedure expects *one* additional continuation argument which it calls at the point it would have originally returned. If a procedure has *two* return points corresponding to *success* and *failure*, then a natural extension of CPS is to pass this procedure *two* continuation arguments: a *success* continuation and a *failure* continuation.

Using this idea write a function that searches a maze for food. A maze is a directed-acyclic graph with nodes defined as follows:

```
(define-record node (id contents left right))
(define-record leaf ())
```

Your procedure should take as arguments: the maze, the item you are looking for (food), and two continuations. If there is food in the maze you should return the path to *some* node containing food. If there is no food, you should return the empty path. For example:

```
(define t1 (make-leaf))
(define t2 (make-node 'n0 'food t1 t1))
(define t3 (make-node 'n1 'water t1 t2))
(define t4 (make-node 'n2 'money t3 t1))

(define init-succ-k (lambda (path) path)
(define init-fail-k (lambda () '()))

(search t1 'food init-succ-k init-fail-k) ==> ()
(search t4 'food init-succ-k init-fail-k) ==> ((n2 . Left) (n1 . Right) n0)
```

Problem 8 (10 points)

Write an evaluator for the following arithmetic expressions (assuming some appropriate abstract syntax). All expressions have their conventional meaning except that any attempt to divide by zero should abort the evaluation and return the string `"Infinity"`.

```
Exp ::= Number | Exp + Exp | Exp - Exp | Exp * Exp | Exp / Exp
```

Hint: Write the evaluator in CPS.

Extra Credit (5 points)

Assuming a statically-scoped, call-by-value language, are the following program transformations correct in all contexts, and for all expressions `M` and `N`?

- Transformation 1:

```
let f = proc (y) N
in begin x:=4; f(3) end
==>
let f = proc (y) begin x:= 4; N end
in f(3)
```

- Transformation 2:

```
f(begin x:=N; M end)
==>
begin x:=N; f(M) end
```

- Transformation 3:

```
begin x:=M; x:=M; N end
==>
begin x:=M; N end
```