CS 565 March 18, 1997

Jens Palsberg

Midterm Exam

Write your name, id number, and department on the top of this page. This exam is intended to take you 2 hours. Budget your time wisely. There are 11 questions on the exam. The first 10 questions are worth 5 points each, and question 11 is worth 10 points, for a total of 60 points maximum. Write your answer on these sheets only. You may use the back side of the paper.

1. Write a Scheme function `add-all-pairs` that takes as arguments two lists of integers l and m, and returns a list which contains all sums of two integers where the first integer is from l and the second integer is from m. The order of the integers in the returned list is implementation dependent (any order is okay!). For example,

```
(add-all-pairs '(1 2) '(30 40)) ==> (31 32 41 42)
(add-all-pairs '(1 2) '())) ==> ()
```

2. Write a Scheme procedure `split` that takes as arguments a list of integers l and an integer i, and returns a list where the first component is a list containing all elements of l which are less than or equal to i, and the rest are the elements of l which are greater than i. For example,

```
(split '(5 3 -2 4 -6) 0) ==> ((-2 -6) 5 3 4)
(split '(1 2 3 4) 1) ==> ((1) 2 3 4)
(split '(1 2 3 4) 5) ==> ((1 2 3 4))
```

3. Consider the following Scheme function:

```
(define mystery
  (lambda (a b)
    (letrec
      ((loop (lambda (c d)
               (cond ((null? c) (loop a d))
                     ((eq? (car c) b) (d b))
                     (else (loop (cdr c) d))))))
      (loop a (lambda (x) x)))))
```

Give two typical examples of calls of `mystery` and the results of the calls, written in the form

```
(mystery ... ...) ==> ...
(mystery ... ...) ==> ...
```

Describe briefly what the procedure `mystery` does.

4. What is produced by the following Scheme program? Justify your answer.

```
(let ((a 1)
      (b (lambda (c a) c))
      (c 4)
      (d 2))
  (let* ((c (+ (b a 2) 3))
         (d (+ 5 c))
         (e (+ d c)))
    (letrec
      ((a (lambda (c) (b e c)))
       (c (lambda (b) (b d)))
       (d (lambda (a) 10)))
      (list
        (a 10)
        (c c))
)))
```

5. Consider this Scheme definition:

```
(define i
 ((lambda (k1)
    ((lambda (k4) (k4 (lambda (x) (lambda (k5) (k5 x)))))
     (lambda (v1)
       ((lambda (k2) (k2 (lambda (y) (lambda (k3) (k3 y)))))
        (lambda (v2) ((v1 v2) k1)))
     )))
  (lambda (a) a)))
```

What is produced by the following two expressions? Justify your answer.

```
> (i 2)
> ((i 2) (lambda (b) b))
```

6. For this problem, you are to modify `interpreter.scm`, which you used in Homework 4. A listing of `interpreter.scm` is on pages 13–20. Write the main code segments on this examination paper, and indicate on the listing where these modifications should be made. You may make small modifications directly on the interpreter listing.

Add to the language the following construction:

$$\langle\text{expression}\rangle ::= \texttt{abs}\ \langle\text{expression}\rangle$$

The construct `abs` e evaluates e. The result is intended to be an integer. The whole expression returns the absolute value of that integer.

7. For this problem, you are to modify `interpreter.scm`, which you used in Homework 4. A listing of `interpreter.scm` is on pages 13–20. Write the main code segments on this examination paper, and indicate on the listing where these modifications should be made. You may make small modifications directly on the interpreter listing.

Add to the language the following constructions:

$$\begin{array}{lcl} \langle\text{expression}\rangle & ::= & \texttt{pair}\ \langle\text{expression}\rangle\ \langle\text{expression}\rangle \\ \langle\text{expression}\rangle & ::= & \texttt{first}\ \langle\text{expression}\rangle \\ \langle\text{expression}\rangle & ::= & \texttt{second}\ \langle\text{expression}\rangle \end{array}$$

In each case, the argument expressions should be evaluated. The construct `pair` $e_1 e_2$ builds a pair of the values of e_1 and e_2. The construct `first` e evaluates e, and if it evaluates to a pair, then the whole expression returns the first component of that pair. The construct `second` e evaluates e, and if it evaluates to a pair, then the whole expression returns the second component of that pair.

8. Convert the following program to tail form:

```
(define pgm
  (lambda (i l)
    (if (eq? (pgm i (cdr l)) (pgm (+ i 1) (cdr (cdr l))))
      #f
      (if (zero? i)
        #t
        (and (pgm (- i 2) l) (pgm (- i 3) (cons (pgm (- i 2) l) l)))))))
```

9. Convert to tail form both of the procedures in following program:

```
(define lau
  (lambda (l m)
    (if (null? (har l))
        m
        (cons (har l) (lau (har (cdr l)) (car l))))))

(define har
  (lambda (l)
    (if (null? l)
        '()
        (cons (lau (cdr l))))))
```

10. Here is a program in tail form. First, on the source program, label all the continuation builders. Then transform it to first-order form by representing all the continuations as records.

```
(define pgm
  (lambda (n)
    (letrec ((loop (lambda (n k)
                     (is-small? n
                       (lambda (v)
                         (if v
                           (k '())
                           (loop (- n 1)
                              (lambda (w)
                                 (k (cons n w)))))))))
             (is-small?
                   (lambda (t k)
                     (k (<= t 0)))))
      (loop n (lambda (x) x)))))
```

11. Here is a program in first-order form. Transform it first into imperative form (5 points), and then translate the result into C code (5 points). (Small syntax errors in the C code are allowed).

```
(define-record id-cont ())

(define-record pgm-cont (i k))

(define pgm
  (lambda (i j)
    (pgm-cps i j (make-id-cont))))

(define pgm-cps
  (lambda (i j k)
    (if (zero? i)
      (apply-cont k (+ j 3))
      (if (zero? j)
          (pgm-cps (- i 2) 2 k)
          (pgm-cps i (- j 2)
            (make-pgm-cont i k))))))

(define apply-cont
  (lambda (k v)
    (record-case k
      (id-cont ()
        v)
      (pgm-cont (i k1)
        (pgm-cps (- i 2) v k1)))))
```

```
;;; Interpreter for Chapter 5/Lecture 4

;;; original: Tue Oct 18 11:02:26 1988
;;; revised Fri Oct 23 10:30:16 1992
;;; revised to use sllgen Tue Sep 28 17:09:43 1993

(printf "interp5.scm Tue Sep 28 17:09:49 1993~%")

;;; Assumes sllgen.scm is loaded

;;; ******************************************************

;;; Top-level interface

(define run
  (lambda (string)
    (eval-exp (scan&parse string) init-env)))

;;; *******************************************************

;;; Lexical Specification

(define automaton-5
  '((proc if then else let set! in =)   ; keywords
    (start-state
      ((#\space #\tab #\newline) #f)
      ((alphabetic  #\* #\+ #\- #\/ #\! #\= #\:)
       (arbno (numeric alphabetic #\* #\+ #\- #\/ #\! #\= #\:))
       identifier)
      ((numeric)
       (arbno numeric)
       number)
      (#\( lparen)
      (#\) rparen)
      (#\^ end-marker)
      (#\% comment-state))
    (comment-state
```

```
        (#\newline #f)
        (any comment-state))))

;;; ****************************************************************

;;; Grammar

(define grammar-5
  '((expression
      (number)
      lit-exp)
    (expression
      (identifier)
      var-exp)
    (expression
      (set! identifier expression)
      assign-exp)
    (expression
      (if expression then expression else expression)
      if-exp)
    (expression
      (let (arbno declaration) in expression)
      let-exp)
    (expression
      (proc lparen (arbno identifier) rparen expression)
      proc-exp)
    (expression
      (lparen expression (arbno expression) rparen)
      app-exp)
  (declaration
    (identifier = expression)
    decl)))

(define scan&parse
  (sllgen:string->tree automaton-5 grammar-5))

;;; End of syntactic specification

;;; ****************************************************************
```

```
;;; Data Structure Definitions for Interpreter

;;; ****************************************************************

;;; Cells

(define make-cell
  (lambda (value)
    (cons '*cell value)))

(define deref-cell cdr)

(define set-cell! set-cdr!)  ; danger!

;;; ****************************************************************

;; Finite functions:  ribcage (list of frames)

; empty-ribcage ==> nil
; (extend-ribcage names vals ff) ==> ((names . vals) . ff)

(define the-empty-ribcage '())

(define extend-ribcage
  (lambda (names vals ribcage)
    (if (= (length names) (length vals))
      (cons (cons names vals) ribcage)
      (error 'extend-ribcage
"wrong number of values. names: ~s values: ~s"
names values))))

(define apply-ribcage
  (lambda (ribcage z)
    (if (null? ribcage)
      (error 'apply-ribcage "identifier ~s not found" z)
      (let ((names (caar ribcage))(vals (cdar ribcage))(f (cdr ribcage)))
(if (memq z names)
  (letrec
```

```
    ;; can assume z will be found in names
    ((loop (lambda (names vals)
     (if (eqv? z (car names)) (car vals)
       (loop (cdr names) (cdr vals))))))
    (loop names vals))
  (apply-ribcage f z))))))

;;; ****************************************************************

;;; Building environments from ribcages:

(define the-empty-env the-empty-ribcage)

(define extend-env
  (lambda (names values env)
    (extend-ribcage names (map make-cell values) env)))

(define apply-env apply-ribcage)


;;; *****************************************************************

;;; Declarations

(define-record decl (var exp))

;;; Closures and procedures

(define-record closure (formals body env))

(define build-user-proc make-closure)


;;; *****************************************************************
;;; *****************************************************************

;;; The Interpreter Proper

(define eval-exp
```

```
  (lambda (exp env)
    (newline)
    (display "eval-exp: exp : ")
    (display exp)
    (newline)
    (record-case exp
      (lit-exp (constant) constant)
      (var-exp (id) (deref-cell (apply-env env id)))
      (assign-exp (ident rhs-exp)
(set-cell!
  (apply-env env ident)
  (eval-exp rhs-exp env)))
      (proc-exp (formals body)
(build-user-proc
  formals
  body env))
      (if-exp (test-exp true-exp false-exp)
(if (zero? (eval-exp test-exp env))
  (eval-exp false-exp env)
  (eval-exp true-exp env)))
      (let-exp (decls body)
(let ((ids (map decl->var decls))
      (exps  (map decl->exp decls)))
  (let ((new-env
  (extend-env ids
    (eval-rands exps env)
    env)))
    (eval-exp body new-env))))
      (app-exp (rator rands)
(let ((proc (eval-exp rator env))
      (args (eval-rands rands env)))
  (apply-proc proc args)))
      (else (error 'eval-exp
      "Bad abstract syntax: ~s" exp)))))

(define eval-rands
  (lambda (rands env)
    (map (lambda (exp) (eval-exp exp env)) rands)))
```

```
(define apply-proc
  (lambda (proc args)
    (record-case proc
      (primitive-proc (primop)
(apply-primop primop args))
      (closure (formals body env)
(eval-exp body
  (extend-env
    formals args env)))
      (else (error 'apply-proc "Bad Procedure ~s" proc)))))


;;; **************************************************************

;;; Primops

(define-record primitive-proc (primop))

(define apply-primop
  (lambda (primop args)
    (case primop
      ((+-op)  (+ (car args) (cadr args)))
      ((--op)  (- (car args) (cadr args)))
      ((*-op)  (* (car args) (cadr args)))
      ((+1-op) (+ (car args) 1))
      ((-1-op) (- (car args) 1))
      (else (error 'apply-primop "Unknown Primop: ~s" primop)))))

;;; **************************************************************

;;; The Initial Environment

(define build-init-env
  (lambda (pairs)
    (extend-env
      (map car pairs)
      (map make-primitive-proc
           (map cadr pairs))
      the-empty-env)))
```

```
(define init-pairs
  '((+ +-op)
    (- --op)
    (* *-op)
    (add1 +1-op)
    (sub1 -1-op)))

(define init-env (build-init-env init-pairs))

;;; ***********************************************************

;;; Tests

(define pgm1 "1")

(define pgm2 "(add1 x)")  ; this one should end on a
; domain error

(define pgm3 "let x = 3 y = 4 in (+ x y)")

(define pgm4 "let f = proc (x) (add1 x) in (f 4)")

(define pgm5 "(proc (x) (add1 x) 4)")

(define pgm6 "let x = 3
              in let y = set! x (add1 x)
                 in x")

(define pgm7 "(+ 3 4)")

; Chez Scheme Version 4.1b
; Copyright (c) 1991 Cadence Research Systems

; > (load "~/.chezrc")
; loading chez-init.s
; finished loading chez-init.s
; > (load "~wand/pub/plangs/sllgen.scm")
```

```
; > (load "interp5.scm")
; interp5.scm Tue Sep 28 17:09:49 1993
; > (run pgm1)
; 1
; > (run pgm2)

; Error in apply-ribcage: identifier x not found.
; Type (debug) to enter the debugger.
; > (run pgm3)
; 7
; > (run pgm4)
; 5
; > (run pgm6)
; 4
; >
; > (run pgm7)
; 7
```