

Name:

ID:

CS 496 A – Programming Languages

Final Exam – Fall 2001

11 December 2001

**This exam is closed book, closed notes, and
closed laptops**

Important Remarks

- This test has 6 questions with a total of 120 points.
- This exam has 20 extra points for your own benefit. The grading scale will remain unchanged: A= 90-120, B= 80-90, C=70-80, D=60-70, else Fail. Therefore the more you complete, the highest your chances to get a higher grade.
- This test is closed notes, closed books and closed laptops.
- The test is timed. We will not grade your test if you try to take more time than allowed. Therefore, before you begin, please take a moment to look over the entire test so that you can budget your time.
- If you find any question unclear, please specify your understanding of what is being asked.

Subset of Scheme You May Use

Unless otherwise stated, when defining procedures you may only use: helping procedures that you define yourself, comments, and the procedures and keywords that are included in the following list. The notation `c..r` means `caar`, `cadr`, `cddr`, etc.

```
' , #t, #f, *, +, -, /, <, <=, =, >=, >,
and, andmap, append, apply, boolean?,
car, cdr, c...r, char? cond, cons,
define, display, else, eq? equal?, eqv?, error, if, let, letrec,
lambda, list, length, list?, map, newline, not, null?, number?,
or, pair?, procedure?, quote,
string, string?, string=?, string-append,
string-ci=?, string-length, string-ref,
string->list, string->number,
string->symbol, substring, symbol?
vector, vector, vector?, vector-length,
vector->list, vector-ref, zero?
define-datatype, cases.
```

1. A 2-slist is a list of lists of symbols of length 2 such as

```
((2 s) (3 x) (y z))
```

```
()
```

- (a) (5 points) Define a BNF for 2-slists
- (b) (10 points) Write a derivation of the 2-slist `((2 s) (3 x) (y z))` in the grammar defined in the previous item.
- (c) (15 points) Define a Scheme procedure `(cartesian-product los1 los2)` that returns a 2-slist that represents the Cartesian product of the lists of symbols `los1` and `los2`. The procedure returns error messages if the inputs are of the wrong type.

```
(define cartesian-product  
  (lambda (los1 los2)
```

```
    ....
```

```
  ))
```

```
; tests
```

```
>(cartesian-product '(1 2) '(a b c))
```

```
((1 a) (1 b) (1 c) (2 a) (2 b) (2 c))
```

```
>(cartesian-product '() '(x y))
```

```
()
```

```
>(cartesian-product '(x y) '())
```

```
()
```

2. Given the following grammar defining the concrete syntax of `<exp>` and the corresponding datatype defining the abstract syntax:

```
<exp> ::= <integer>
        | <boolean>
        | <identifier>
        | (if <exp> <exp> <exp>)
        | (and <exp>+)
        | (or <exp>+)
        | (not <exp>)
```

```
(define-datatype exp exp?
  (lit-exp (datum number?))
  (bool-exp (bool boolean?))
  (var-exp (var symbol?))
  (if-exp (test exp?)
          (then exp?)
          (else exp?))
  (and-exp (args (list-of exp?)))
  (or-exp (args (list-of exp?)))
  (not-exp (arg)))
```

(20 points) write a Scheme procedure `parse` that transforms a concrete syntax expression into its abstract syntax tree.

```
(define parse
  (lambda (datum)
    (cond
      ....
    )))

; tests
>(parse 'a)
(var-exp a)

>(parse 1)
(lit-exp 1)

>(parse #t)
(bool-exp #t)

>(parse #f)
(bool-exp #f)

>(parse '(or 3 4 3))
(or-exp ((lit-exp 3) (lit-exp 4) (lit-exp 3)))
```

3. (5 points) What is wrong with the following expression?

```
let x = 4
    y = 8
    z = +(x, 2)
in -(y, z)
```

4. (15 points) Trace the following program under the call-by-value parameter passing style.

```
let a = 19
    b = 5
in let p = proc (x, y)
    let t = x
        u = y
    in begin
        set x = y;
        set y = t;
        set a = u
    end
in begin
    (p a b);
    -(a, b)
end
```

5. Consider the following grammar for the language of programs and statements, where expressions are left unspecified for the purpose of this exercise.

```
<program> ::= <statement>

<statement> ::= <identifier> = <expression>
| print (<expression>)
| {{<statement>}*(&#92;)}
| if <expression> <statement> <statement>
| while <expression> do <statement>
| var {{<identifier>}*(&#92;) ; <statement>
```

Where `{{<statement>}*(\)}` is a sequence of statements separated by `'\'`, and similarly for `{{<identifier>}*(\)}`

- (a) (10 points) Extend the language of statements with a read statement of the form `read (<identifier>)` and add the corresponding entry to the interpreter. This statement reads a non-negative integer from the input and stores it in the given variable.
- (b) (20 points) A *do-while* statement is like a while statement except that the test is performed *after* the execution of the body (the body is always executed at least once). Add an entry for *do-while* statements to the interpreter.

```

;;;;;;; Interpreter ;;;;;;;;

(define execute-program
  (lambda (pgm)
    (cases program pgm
      (a-program (statement)
        (execute-statement statement (init-env))))))

(define execute-statement
  (lambda (stmt env)
    (cases statement stmt
      (assign-statement (id exp)
        (setref!
          (apply-env-ref env id)
          (eval-expression exp env)))
      (print-statement (exp)
        (write (eval-expression exp env))
          (newline))
      (compound-statement (statements)
        (for-each
          (lambda (statement)
            (execute-statement statement env))
          statements))
      (if-statement (exp true-statement false-statement)
        (if (true-value? (eval-expression exp env))
            (execute-statement true-statement env)
            (execute-statement false-statement env)))
      (while-statement (exp statement)
        (let loop ()
          (if (true-value? (eval-expression exp env))
              (begin
                (execute-statement statement env)
                (loop))))))
      (block-statement (ids statement)
        (execute-statement statement
          (extend-env ids (map (lambda (id) 0) ids) env))))

    ...

  )))

```

6. (a) (15 points) Compute the type of the following `lettype` expression. Justify your answer.

```
lettype ff = (int -> int)
  ff zero-ff () = proc (int k) 0
  ff extend-ff (int k, int val, ff old-ff) =
    proc (int k1)
      if zero? (- (k1,k))
      then val
      else (apply-ff old-ff k1)
  int apply-ff (ff f, int k) = (f k)
in let ff1 = (extend-ff 1 11
              (extend-ff 2 22
                (zero-ff)))
  in (apply-ff ff1 2)
```

- (b) (5 points) What is wrong with the following expression?

```
lettype ff = (int -> int)
  ff zero-ff () = proc (int k) 0
  ff extend-ff (int k, int val, ff old-ff) =
    proc (int k1)
      if zero? (- (k1,k))
      then val
      else (apply-ff old-ff k1)
  int apply-ff (ff f, int k) = (f k)
in let ff1 = (extend-ff 1 11
              (extend-ff 2 22
                (zero-ff)))
  in (ff1 2)
```