

Name:

ID:

CS 496 A – Programming Languages

Midterm Exam – Fall 2001

16 October 2001

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

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. 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>)
        | (lambda <exp>* <exp>)
        | (and <exp>+)
        | (<exp> <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?))
  (lambda-exp (formals (list-of symbol?))
              (body exp?))
  (and-exp (args (list-of exp?)) )
  (app-exp (rator exp?)
           (rands (list-of exp?))) )
```

- (a) (10 points) write a derivation using the rules of the grammar defining `<exp>`, of the concrete syntax expression
(lambda (x y z) (if x (y x x) (z x x)))
- (b) (30 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 '(lambda (x y z) (if x (y x x) (z x x))))
  (lambda-exp
   (x y z)
   (if-exp
    (var-exp x)
    (app-exp (var-exp y) ((var-exp x) (var-exp x)))
    (app-exp (var-exp z) ((var-exp x) (var-exp x)))))

>(parse '(and 3 4 3))
  (and-exp ((lit-exp 3) (lit-exp 4) (lit-exp 3)))

```

2. (20 points) Let $e \in \langle \text{bool-exp} \rangle$ where $\langle \text{bool-exp} \rangle$ is defined as follows:

```

<bool-exp> ::=
  | <boolean>
  | <identifier>
  | if <bool-exp> then <bool-exp> else <bool-exp>

```

Then e contains an equal number of if, then and else keywords.

Hint: The induction principle on $\langle \text{bool-exp} \rangle$ for $e \in \langle \text{bool-exp} \rangle$ says that

If

- (a) $P(b)$ for b in $\langle \text{boolean} \rangle$
- (b) $P(id)$ for id in $\langle \text{identifier} \rangle$
- (c) If $P(e_1)$, $P(e_2)$, and $P(e_3)$ hold then $P(\text{if } e_1 \text{ then } e_2 \text{ else } e_3)$

Then $P(e)$ for every $e \in \langle \text{bool-exp} \rangle$.

3. (20 points) Define a Scheme procedure (`make-three-copies lst`) that returns a list where each element of `lst` appears three times in succession.

```
(define make-three-copies
  (lambda
    ....

  ))

; tests
>(make-three-copies '(1 2 3))
(1 1 1 2 2 2 3 3 3)
>(make-three-copies '((foo bar) (boof baz)))
((foo bar) (foo bar) (foo bar) (boof baz) (boof baz) (boof baz))
>(make-three-copies '())
()
```

4. (20 points) Write a Scheme procedure (`zip l1 l2`) that takes two lists and “zips” them together to obtain a list of pairs with components from the corresponding list. You may assume that the lists `l1` and `l2` are of equal length.

```
(define zip
  (lambda (l1 l2)
    ...
  ))

;test

>(zip '(1 2 3) '(4 5 6))
((1 . 4) (2 . 5) (3 . 6))

>(zip '() ())
()

>(zip '(1) '(1))
((1 . 1))

>(zip '(posh car sporty scary) '(nutmeg pepper basil oregano))
((posh . nutmeg) (car . pepper) (sporty . basil) (scary . oregano))
```