University of Toronto at Mississauga—Erindale College
csc324—Principles of Programming Languages, Winter 2001

# Midterm Test Solutions

## March 5, 2001

---

**Duration:** 90 minutes

**Aids allowed:** None

**Weight:** 20% of your course grade

This exam contains a total of 9 pages (including this one). Write your answers clearly in the spaces provided. Use the back pages for your rough work.

**Surname:** _____

**First name:** _____

**Student #:** _____

# 0: _____/ 2
# 1: _____/ 5
# 2: _____/ 4
# 3: _____/ 6
# 4: _____/ 3
# 5: _____/ 6
# 6: _____/ 6
# 7: _____/22
# 8: _____/ 5
# 9: _____/ 6
# 10: _____/10

TOTAL: _____/75

Good Luck!

## Question 0. [2 MARKS]

Write your name (or your initials if your name is long) and student number legibly at the top of every page of this test.

## Question 1. List Manipulation [5 MARKS]

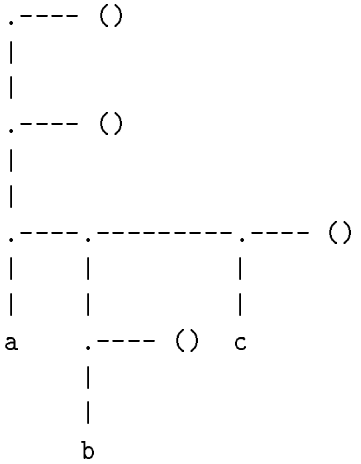
Each of the following expressions has a missing piece. Give the name of one Scheme function that completes the expression, making it have the value shown. Your answer should be one of `cons`, `list`, `append`, `car`, or `cdr`; or write "cannot" if none of these functions causes the expression to have the value shown.

| Expression | Desired value | function name that completes the expression (or "cannot") |
|---|---|---|
| `(cons (car '(1 2)) (  '(3 4)))` | `(1 3)` | cannot: car returns 3, not (3) |
| `(  (1 2) (3 4))` | `((1 2) (3 4))` | cannot: need quotes on the lists |
| `(cons (  '((3 4) 3 4)) '(1 2))` | `(3 4 1 2)` | cannot: need append instead of cons |
| `(append (car '(1 2)) (  '(1 2 3 4)))` | `(1 2 3 4)` | cannot: need cons instead of append |
| `(  '() '(cdr (() 2 3)))` | `(() 2 3)` | cannot: quote in the wrong place |

## Question 2. Memory Representation of Lists [4 MARKS]

Draw the memory representation of the following list:

`(((a (b) c)))`



## Question 3. Function Calling Mechanisms in Scheme [6 MARKS]

For each of the expression below, write what it will output if it is correct, or briefly say why it is wrong.

| | |
|---|---|
| `(let ((a 1)) (apply + '(a 3)))` | 4 |
| `(let ((a 1)) (eval '(+ a 3) ()))` | `a` not bound |
| `(let ((a 1)) (+ '(a 3)))` | wrong type of arg for `+` |
| `(let ((a 1)) (apply + a 3))` | too many args for apply |
| `(let ((a 1)) (eval '(+ (a 3)) ()))` | `a` unbound *or* wrong type of arg for `+` |
| `(let ((a 1)) (+ a 3))` | 4 |

## Question 4. Using HOFs [3 MARKS]

Each of the following expressions has a missing piece. Fill in the blank space with one of `map`, `apply`, `eval`, `prune` (as seen in class), or `reduce` (as seen in class), making it have the value shown, or write "cannot" if none of these causes the expression to have the desired value.

| Expression | Desired value |
|---|---|
| `(` *cannot* `reverse '((1 2) (3 4)))`<br>*or* state "should be left blank" to call `reverse` directly | `((3 4) (1 2))` |
| `(` `map` `reverse '((1 2) (3 4)))` | `((2 1) (4 3))` |
| `(` *cannot* `reverse '((1 2) (3 4)))`<br>*or change to* `(map reverse (reverse ... ))`<br>*or* `(reverse (map reverse ... ))` | `((4 3) (2 1))` |

## Question 5. Writing Scheme functions [6 MARKS]

Write a function (odd-elements lst) which returns a list with the elements of `lst` in the odd positions. (Assume the list has no sublists.) For example,

```
(odd-elements '())           returns  ()
(odd-elements '(a b c d))    returns  (a c)
(odd-elements '(a b c d e))  returns  (a c e)
```

Solution:

```
(define (odd-elements lst)
    (cond ((null? lst) ())
          ((null? (cdr lst)) lst)
          (else (cond (car lst) (odd-elements (cddr lst))))
    )
)
```

## Question 6. Writing Scheme functions [6 MARKS]

Write a function `(deep-count x lst)` which counts how many times `x` occurs in `lst`. For example,

```
(deep-count 'a '(a b c b a))          returns  2
(deep-count 'a '())                   returns  0
(deep-count 'a '(a b (a a) ((a)) b))  returns  4
```

Note: you **must** use `map` in your solution. You may also use other higher-order functions if you need them.

Solution:

```
(define (deep-count x lst)
    (apply + (map (lambda (y) (cond ((list? y) (deep-count x y))
                                    ((eq? x y) 1)
                                    (else 0)))
                  lst))
)
```

## Question 7. Syntax [22 MARKS]

### Part (a) [4 MARKS]

Describe in English the language that is generated by the following BNF grammar.

```
<statement>  -->  B <statement> | Z <plunger>
<plunger>    -->  Z <plunger> | F <pretzel>
<pretzel>    -->  S <pretzel> | S
```

Solution: All strings containing 0 or more B's, followed by 1 or more Z's, then one F, and then one or more S's. Can also be stated with the regular expression $B^*Z^+FS^+$.

### Part (b) [4 MARKS]

Prove that this grammar is ambiguous:

```
<E>  -->  <E> - <E> | 0 | 1
```

Solution: Show two parse trees for, e.g., `0 - 0 - 0`, and conclude that the grammar is ambiguous.

### Part (c) [4 MARKS]

Transform the grammar from question (b) into a grammar that is unambiguous but that accepts the same strings. Use BNF notation. Note that there are several correct answers.

Solution:

```
<E>  -->  <E> - <N> | <N>
<N>  -->  0 | 1
```

## Question 7. Syntax (CONTINUED)

### Part (d) [6 MARKS]

Consider the following BNF grammar:

```
<pop>  --> + <bop> , <pop> = | <bop>
<bop>  --> <boop> | (<pop>)
<boop> --> x | y | z
```

Which of the strings below is a `<pop>`? Circle yes or no as appropriate.
Do not guess; correct answers are worth 1 mark, and incorrect answers are worth -1 mark.

| | Is it a `<pop>`? | |
|---|---|---|
| `z` | [Yes] | No |
| `(x)` | [Yes] | No |
| `+y=` | Yes | [No] |
| `(+y=)` | Yes | [No] |
| `+(x),y=` | [Yes] | No |
| `+(x),+y,x==` | [Yes] | No |

### Part (e) [4 MARKS]

Give an unambiguous context-free grammar in BNF that can generate the language

$$a^{2n}b^m cd^n, n \geq 0, m > 0.$$

```
<S>  -->  aa <S> a | <B> c
<B>  -->  b <B> | b
```

## Question 8. Type Checking [5 MARKS]

Assume the assignment statements below are syntactically correct in some imperative language. The necessary type definitions are given as "type-name = description" and variable declarations are given as "type: var-name". Full array, structure, and string assignments are allowed in this language.

```
type VOWELS = (a, e, i, o, u);
     WORDS = array[VOWELS] of string;
     LENGTH = 1..26;
     PHRASE = struct { 1..26: subject;
                       1..26: verb;
                       WORDS: rest; };

var  array[1..100] of WORDS: sentences;
     array[a..i] of string: letters;
     VOWELS: v;
     LENGTH: n;
     integer: i;
     PHRASE: p;
```

For each statement below, write C (for compile time) next to those that are statically type checkable, and R (for run time) next to those that are only able to be type checked dynamically.

1. R `read(i);`
2. R `p.subject := i;`
3. R `v = succ(v);`
4. R `letters[v] := "letter";`
5. C `sentences[n][v] := "sent";`

## Question 9. Type Equivalence [6 MARKS]

Assume some hypothetical imperative language uses the following type equivalence rules:

1. A type name is equivalent to itself.
2. Two types are equivalent if they are formed by applying the same type constructor to equivalent types.

Below are several type and variable declarations in this language:

```
type intarray = array[0..9] of integer;
     i = integer;

var  a: i;
     b: intarray;
     c: array[0..9] of integer;
     d: array[0..9] of integer;
     e: array[0..9] of i;
     f: integer;
```

### Part (a) [3 MARKS]

Indicate which of these variables are equivalent under the type equivalence rules of the hypothetical imperative language mentioned above.

Solution:
Only the following variables have equivalent types:

- `c`, `d`.

### Part (b) [3 MARKS]

Indicate which of these variables are equivalent under declaration equivalence.

Solution:
The following variables have declaration equivalent types:

- `a`, `f`.

## Question 10. Short Questions [10 MARKS]

Note: only short and concise answers will be graded. Long-winded answers will get a mark of 0.

**(a)** [2 MARKS] What characterizes imperative programming?

Assignments and loops; statements are commands.

**(b)** [2 MARKS] Give two properties of well-written grammars.

Any two of captures the logical structure of the language, uses meaningful names, easy to read, unambiguous, etc.

**(c)** [2 MARKS] Name one advantage of explicit typing over implicit typing, and one advantage of implicit typing over explicit typing.

Explicity typing supports static type determination. Implicit typing gives more flexibility to the programmer.

**(d)** [2 MARKS] What is a type-safe program?

A program that executes without type errors on all possible inputs.

**(e)** [2 MARKS] What is a strongly typed programming language?

A language whose definition forces all type-checked programs to be type safe.

Total Marks = 75