CSC C24 Winter 2023
Midterm Test
Duration — 90 minutes
Aids allowed: none

*Do **not** turn this page until you have received the signal to start.*

Please fill out the identification section above and read the instructions below.

Good Luck!

This exam is double-sided, and consists of 3 questions. *When you receive the signal to start, please make sure that your copy is complete.*

- Please, make sure to **NOT write anything in the QR code areas**.
- Please, use a **black** or **blue pen** or a **thick in diameter pencil** to answer **all questions** in this booklet.
- Comments are **not** required except where indicated, although they may help us mark your answers.
- No error checking is required: assume all user input and all argument values are **valid**.
- Please, make sure to **indent** your code properly.
- If you use any space for rough work, indicate clearly what you want marked.

Question 1. [10 MARKS]

Consider the following part of a CFG for a programming language. We will assume that the rules for `<boolean-expr>`, `<assignment-stmt>`, and `<loop-stmt>` have been designed correctly. Just as in your assignment, the semicolon `;` is used for sequential composition of statements. The start symbol is `<stmts>`.

```
<stmts>    --> <assignment-stmt> | <loop-stmt> | <if-stmt> | <stmts> ; <stmts>
<if-stmt> --> if <boolean-expr> then <stmts>
           |  if <boolean-expr> then <stmts> else <stmts>
```

There are two serious problems with this grammar: one is related to sequential composition and another to the if-statements.

Part (a) [5 MARKS]

Show how to fix the problem with sequential composition by making it **left-associative**.

Part (b) [5 MARKS]

Demonstrate that the if-statement causes ambiguity by producing two different parse trees for a string in the language. Since you don't know what `<boolean-expr>`, `<assignment-stmt>` and `<loop-stmt>` generate, you can leave them as leafs in your parse trees. Use the original grammar to generate the trees.

Question 1. (CONTINUED)

Question 2. [10 MARKS]

As we have seen in class, there is a number of ways in which we can specify function parameters in Racket:

- no brackets (Racket will bind parameter to list of arguments)
  - `(define my-func param some-expr )`
- no parameters
  - `(define my-func () some-expr )`
- one or more parameters
  - `(define my-func (param0 param1 ...  paramN ) some-expr )`
- two or more parameters with a single "." somewhere after the first and before the last parameter
  - `(define my-func (param0 .  param1 ...  paramN) some-expr )`
  - `(define my-func (param0 param1 .  param2 ...  paramN) some-expr )`
  - etc.

where `some-expr` contains the body of the function

I started writing a CFG for these definitions. Luckily, someone already developed production rules for `<some-expr>` that generates all function bodies and for `<ident>` that generates all identifiers in the language (including function names, parameter names, etc.), so I can use the non-terminals `<some-expr>` and `<ident>` in my rules. Help me finish the grammar, making sure it generates all required strings as is not ambiguous.

Terminals: ____

plus everything generated by <some-expr>

Non-terminals: ____

____

Start symbol: ____

Production rules:

<definition> ::=

Question 2. (continued)

Question 3. [30 MARKS]

Complete the implementations of the following functions. Note that **implementations that do not follow the requirements in the comments will not earn any marks**.

Part (a) [5 MARKS]

```
;; (zip-rec xs ys) -> list?
;; xs, ys: list?
;; Returns a list of pairs of corresponding elements in xs and ys.
;; Remaining elements in the longer list are ignored.
;; This is a recursive implementation.
(define (zip-rec xs ys)
```

```
(check-expect (zip-rec '(1 2 3) '()) '())
(check-expect (zip-rec '() '(1 2 3)) '())
(check-expect (zip-rec '(1 2 3) '(a b c d)) '((1 . a) (2 . b) (3 . c)))
```

Part (b) [3 MARKS]

```
;; Same as zip-rec above, but non-recursive, uses a single call to map, and no other
;; higher order procedures. In this implementation, assume xs and ys are of the same size.
(define (zip-map xs ys)

  (map ____________________________________________________________

       ____________________________________________________________

       ____________________________________________________________ ))
```

Part (c) [5 MARKS]

```
; (unzip pairs) -> list?
; pairs: list of pairs
; Returns the result of unzipping pairs, namely, a list of two lists:
; the first sublist is a list of first elements of all pairs, in order,
; and the second sublist is a list of second elements.
; This is a recursive implementation.
(define (unzip pairs)
```

```
(check-expect (unzip '()) '(() ()))
(check-expect (unzip '((42 . 24))) '((42) (24)))
(check-expect (unzip '((1 . -1) (2 . -2) (3 . -3) (4 . -4)))
              '((1 2 3 4) (-1 -2 -3 -4)))
```

Part (d) [4 MARKS]

```
; This implementation uses map and no recursion.
(define (unzip-hop pairs)
```

Part (e) [5 MARKS]

```
; This implementation uses foldr and no recursion.
(define (unzip-fold pairs)
  (foldr
```

Part (f) [4 MARKS]

```
;; (test-apply ok? f g xs) -> list?
;; ok? : procedure?
;; f : procedure?
;; xs : list?
;; Return the list of results obtained as follows: for each element x in xs,
;; apply f if (ok? x) holds, and apply g otherwise.
;; This implementation is not recursive and uses a single call to map and no other
;; higher-order functions.
(define (test-apply ok? f g xs)
  (map
```

```
(check-expect (test-apply positive? (lambda (x) (* 2 x)) (lambda (x) (- x 10)) '())
              '())
(check-expect (test-apply positive? (lambda (x) (* 2 x)) (lambda (x) (- x 10))
                          '(-1 2 -3 4))
              '(-11 4 -13 8))
```

Part (g) [4 MARKS]

```
;; This implementation is not recursive and uses a single call to foldr and no other
;; higher-order functions.
(define (test-apply-f ok? f g xs)
  (foldr
```