PLEASE HAND IN

PLEASE HAND IN

UNIVERSITY OF TORONTO
Faculty of Arts and Science

APRIL 2017 EXAMINATIONS

CSC 324 H1S
Instructor(s): G. Baumgartner

Duration — 3 hours

No Aids Allowed

Student Number: |_|_|_|_|_|_|_|_|_|_|

Last (Family) Name(s): ____________________

First (Given) Name(s): ____________________

---

*Do **not** turn this page until you have received the signal to start.*
In the meantime, please read the instructions below *carefully*.

You must receive at least 40% on this exam to pass the course.

This Final Examination paper consists of 9 questions on 20 pages (including this one), printed on both sides of the paper. *When you receive the signal to start, please make sure that your copy of the paper is complete and fill in your name and student number above.*

Answer each question directly on this exam paper, in the space provided. If you need more space for one of your solutions you may also use a "blank" page at the end of the paper (in which case make sure to mention that where the question is asked).

If you leave a Question blank, or a Part of a Question blank, or clearly cross out your answer with a diagonal line, you will receive 25% of the marks allocated to that Question or Part.

After the last question there are four pages of reminders for the racket and typed/racket language.

MARKING GUIDE

# 1:	______/10
# 2:	______/ 5
# 3:	______/ 5
# 4:	______/ 5
# 5:	______/ 5
# 6:	______/15
# 7:	______/15
# 8:	______/15
# 9:	______/10
TOTAL:	______/85

Good Luck!

Question 1. [10 MARKS]

Part (a) [4 MARKS] Show the values resulting from evaluating the following two expressions.

```
(apply append (list (list (list 1 2) (list 3 4))
                    (list (list 5 6) (list 7 8))))
```

'((1 2) (3 4) (5 6) (7 8))

```
(map append
     (list (list 1 2) (list 3 4))
     (list (list 5 6) (list 7 8)))
```

'((1 2 5 6) (3 4 7 8))

Part (b) [6 MARKS] Consider the following definitions of functions R and RA.

```
(define (R LoL)
  (cond [(list? LoL) (reverse (map R LoL))]
        [else LoL]))

(define (RA LoL)
  (cond [(list? lol) (reverse (apply append (map RA LoL)))]
        [else (list LoL)]))
```

Show the values resulting from evaluating the following two expressions.

```
(R (list (list 1 (list 2 3))))
```

'(((3 2) 1))

```
(RA (list (list 1 2) 3 4))
```

'(4 3 1 2)

Question 2. [5 MARKS]

List all the FUNCTION CALL expressions that appear in the following expression.

```
(a (b (c d)) (e f g) h (i (j)))
↑  ↑  ↑      ↑         ↑  ↑
```

List the names of the functions that will be called, in the order that they will be called.

c b e j i a

Question 3. [5 MARKS]

Consider the following function definition:

```
(define (add-up n)
  (if (positive? n)
      (+ n (add-up (sub1 n)))
      0))
```

List all the sub-expressions in the body of add-up that are in tail position with respect to the body.

Question 4. [5 MARKS]

Define sub-parts to produce a list containing its argument, all the elements of its argument, and so on, recursively.

```
(check-expect (sub-parts '(((1)) (2 3) 4)) '((((1)) (2 3) 4)
                                             ((1))
                                             (1)
                                             1
                                             (2 3)
                                             2
                                             3
                                             4))
```

```
(define (sub-parts v)
  (list* v
     (cond [(list? v) (apply append (map sub-parts v))]
        [else '()])))
```

Question 5. [5 MARKS]

Consider the following definitions of implies and sorted?.

```
#| implies : (Boolean Boolean → Boolean) |#

(define (implies b0 b1) (or (not b0) b1))      b0 is false or b1 is true

#| sorted? : ((Listof Real) → Boolean)

 Whether list of real numbers L is in increasing order.
 Empty lists and lists with a single element are considered sorted. |#

(define (sorted? L)
  (implies (>= (length L) 2)
           (and (< (first L) (second L))
                (sorted? (rest L)))))
```

Give the simplest test case for which sorted? fails, and state exactly how it fails.

'()
calling implies requires evaluating argument expression,
calling first on empty list is invalid

Question 6. [15 MARKS]

Consider the following code.

```
(define (Counter i)
  (λ (d)
    (set! i (+ d i))
    i))

(define c (Counter 0))

(map c '(1 20))

(map (λ (i) ((Counter i) i)) '(300 4000))
```

Show the memory model after evaluating that code.
Include:

- the tree of environments
- each closure, pointing to its environment either from outside the tree or interleaved

State the resulting value of each of the two top-level non-definition expressions.

Question 7. [15 MARKS]

Recall the backtracking choice operation `-<`, aka "either".
The form of its use is: `(-< <expression> ...)` .

Recall the `fail` operation.
It aborts the most recently chosen expression and backtracks/continues with the next waiting expression.
The form of its use is: `(fail)` .

Part (a) [5 MARKS]

Define an operation `when-assert` that takes a condition expression and body expressions.
The form of its use is: `(when-assert <condition-expression> <body-expression> ...)` .
If the condition evaluates to true, it produces the result of evaluating the body expressions in order, otherwise it fails.
Use only `-<` and/or `fail` for the backtracking behaviour.

Part (b) [5 MARKS]

Define an operation `cond-<` that has clauses similar to `cond`, except there is no special `else` clause.
It chooses, in the sense of the choice operator, to evaluate the result expressions from one of the clauses for which the condition is true.
Use only `-<` and/or `when-assert` for the backtracking behaviour.
The form of its use is:

```
(cond-< [<condition-expression> <result-expression> ...]
        ...)
```

For example,

```
(cond-< [(< 1 2) 3]
        [(> 1 2) (println 'hi) 4]
        [(zero? (random 2)) (println 'totally) 5])
```

would produce 3, and if backtracked would possibly print `'totally` and produce 5.

Part (c) [5 MARKS]

Define backtracking function `generate-term` to produce a term from the language Term defined by:

- Symbols `sin`, `cos`, and `exp` are in Term.
- If t_0 and t_1 are in Term, then so are the lists $(t_0 + t_1)$, $(t_0 \times t_1)$, and $(t_0 \circ t_1)$.

More specifically: `(generate-term recursions)` produces a base-case term if `recursions` is zero, otherwise it produces a term whose two component terms are each built using exactly one less recursive step.

Use only `-<` and/or `cond-<` for the choice behaviour, do not use any other conditionals.

Use the choice operation(s) as locally as possible to a choice being made, to avoid code duplication.

Use quasi-quotation to build compound terms.

```
; generate-term : (Natural → Term)
```

Question 8. [15 MARKS]

This question is in `#lang typed/racket` .
Recall our compile-time definition of the `Stack` type, as an alias for writing `(Listof Real)`.

```
(define-type Stack (Listof Real))
```

Recall our polymorphic struct type `stack-result`.

stack-result parameterized by alpha

```
(struct (α) stack-result ([stack : Stack] [result : α]) #:transparent)
```

Consider this definition of a function `lift`:

```
(: lift : (∀ (α β) ((α → β) → (α → (Stack → (stack-result β))))))
(define (((lift f) a) s)
  (stack-result s (f a)))
```

Part (a) [3 MARKS]

Implement `number-of-items` according to its static type and example use.

```
(: number-of-items : (Stack → (stack-result Natural)))

(check-equal? (number-of-items '(324 263 209 236 258 207))
              (stack-result '(324 263 209 236 258 207) 6))
```

```
(define (number-of-items stack)
  (match stack
    [`(,e ...) (stack-result stack (length e))]))
```

Part (b) [3 MARKS]

Give an equivalent definition of `lift`, using only the `(define <identifier> <expression>)` form of `define`.

```
(define lift′
  (λ (f)
    (λ (a)
      (λ (s)
        (stack-result s (f a))))))
```

Part (c) [3 MARKS]

Here is the type of `println`, as a compile-time assertion, along with a definition of `g` using it.

```
(ann println (Any → Void))
(define g (lift println))
```

Declare the type of `g`. basically, substitute Any and Void to type of lift

```
(: g :
```

(: g : (Any → (Stack → (stack-result Void))))

Part (d) [3 MARKS]

Write a valid (at compile-time and run-time) expression involving a call of `g`, and show the type of the expression as a compile-time assertion, in the form: `(ann <expression-calling-g> <type-of-the-expression>)` .

; compile time assertion for a call of g
(ann (g 1) (Stack → (stack-result Void)))

Part (e) [3 MARKS]

Write an expression that uses `g` to produce a `stack-result`.
State the side-effects and result value for your expression.

((g 'calling-g) '(0 1 2 3))
; side-effect: 'calling-g symbol is printed

Question 9. [10 MARKS]

Part (a) [4 MARKS]

Declare the (Hindley-Milner) non-polymorphic static type of `f`.

```
#lang typed/racket

(: f :


(define (f x y)
  (if (= 324.5 (first x))
      y
      "Hello!"))
```

Part (b) [3 MARKS]

Consider the function g.

```
(define (g a b c)
  (if (empty? b) '()
      (if (empty? c) '()
          (list* (a (first b) (first c))
                 (g a (rest b) (rest c))))))
```

Write an illustrative test case for g.

Part (c) [3 MARKS]

Declare the (Hindley-Milner) polymorphic static type of g.

```
#lang typed/racket

(: g : (∀ (          )
```

Booleans, Numbers, Strings, Symbols.

Literal	Predicate	Static type
#true	boolean?	Boolean
1/2	number?	Real
"string"	string?	String
'symbol	symbol?	Symbol

Closures.

```
(λ/lambda (<id> ...) ; Fixed arity.
          <body-expr/defn>
          ...)
```

Predicate: procedure?

Static Types:

```
 (<type> ... → <result-type>)
 (<type> * → <result-type>)
```

Calling: (<function-expression> <argument-expression> ...)

Naming.

```
(define <id> <expr>)

(define (<function-id> <parameter-id> ...)
  <body-expr/defn>
  ...)

(define ((<function-id> <parameter-id> ...) <parameter-id> ...)
  <body-expr/defn>
  ...)

; etc

(local [<definition>
        ...]
  <body-expr>
  ...)
```

Variable Mutation: (set! <id> <expr>)

```
Conditional Evaluation.

  (and <expr> ...)
  (or <expr> ...)

  (if <condition-expr> <consequent-expr> <alternative-expr>)

  (cond [<condition-expr> <expr/defn> ...]
        ...)

  (cond [<condition-expr> <expr/defn> ...]
        ...
        [else <else-expr/defn> ...])

  (when <condition-expr>
    <expr/defn>
    ...)

  (unless <condition-expr>
    <expr/defn>
    ...)

Quotation.

  '<id>  '<literal>
  '(<part> <...>) : <...> means multiple <parts> allowed, vs literally  ... .
  '(<first> . <rest>)

  Quasi-quotation:  `  instead of  '  , escape via  ,  .

Runtime Pattern Matching.

  (match <expr> [<pattern> <expr/defn> <...>]
                <...>)

  Patterns:
    <id>
    Literal and quasiquoted <pattern>s.
    ...
    (<struct-id> <pattern> <...>)
    (list <pattern> <...>)
    (list* <pattern> <...>)

Immutable Struct Declaration.

  (struct <struct-id> (<field-id> ...) #:transparent)
  <struct-id> : constructor
  <struct-id>? : predicate
  <struct-id>-<field-id> : field accessor
```

```
User-Defined Syntactic Forms / Macros.

  Single pattern-template, no literals: (define-syntax-rule <pattern> <template>)

  Multiple pattern-template clauses and/or literals:

    (define-syntax <id> (syntax-rules (<literal-id> <...>)
                          [(<pattern> <template>)]
                          <...>))

  Use of  ...  in <pattern> and <template>.

Continuations.
  prompt : delimit continuations for abort and let-continuation
  abort  : abort to the most recent dynamically entered prompt
  Name the continuation to the most recent dynamically entered prompt:
    (let-continuation <id>
      <expr/defn>
      ...)

Backtracking.
  Implement code that backtracks:
    (-< <expr> ...)
    (fail)
  Use code that backtracks:
    (results <expr>)
    (next)
```

```
Static Types.

  Atomic: Void Boolean Real String Symbol Natural Any

  List types: (Listof <type>)

  Function types:
    (<type> ... → <result-type>)
    (<type> * → <result-type>)

   Struct types: <struct-id>

   Union types: <union-id>

   Create a new kind of struct type:

     (struct <struct-id> ([<field-id> : <type>] ...) #:transparent)

   Create and name a union of struct types: (define-type <union-id> (U <struct-id> ...))

   Define a compile-time alias for a type: (define-type <id> <type>)

   Declare that <id> has static type <type>: (: <id> : <type>)

   Compile-time assertion that <expr> conforms to <type>, and treat it that way: (ann <expr> <typ

   Polymorphic/Universal Types: (∀ (<type-parameter-id> ...) <type>)

   Create a polymorphic struct type:
     (struct (<type-parameter-id>) <struct-id>  ([<field-id> : <type>] ...)
        #:transparent)

   Specific version of a polymorphic struct type: (<struct-type> <type-argument>)
```

OVER...

Use the space on this "blank" page for scratch work, or for any answer that did not fit elsewhere.
Clearly label each such answer with the appropriate question and part number.

CONT'D...

Use the space on this "blank" page for scratch work, or for any answer that did not fit elsewhere.
Clearly label each such answer with the appropriate question and part number.

Please write nothing on this page.