Compilers (Racket)
CSCI P423/523, Fall 2022

Midterm

Name: ____________________

This exam has 9 questions, for a total of 100 points.

1. 10 points Given the grammar below for expressions, indicate which of the following programs are in the language specified by the grammar. That is, which can be parsed as the *exp* non-terminal.

```
exp ::= int | (read) | (- exp) | (if exp exp exp) | var | (eq? exp exp)
      | (let ([var exp]) exp)
```

1. `(- (if (eq? (read) 0) 10 20))`

2. ```
   (let ([x (if (eq? (read) 0) (- 10))])
     (+ x 10))
   ```

3. `(eq? (- (- (- (read)))) (- (- (- 10))))`

4. `(let ([x (if (eq? (read) 0) 10 (- 10))]))`

5. `(- (read))`

Solution: 2 points each

1. Yes.
2. No, one-armed `if` is not an *exp*.
3. Yes.
4. No, the `let` is incomplete, missing its body.
5. Yes.
   ```

2. $\boxed{\text{12 points}}$ Convert the following program to its Abstract Syntax Tree representation (see the grammar for $\mathcal{L}_{\mathsf{If}}$ in the Appendix of this exam) and draw the tree.

```
(let ([x (if (eq? (read) 0) 5 (- (read)))])
   (+ x 42))
```

Solution:

```
(Program '()
 (Let 'x
  (If (Prim 'eq? (list (Prim 'read '()) (Int 0)))
      (Int 5)
      (Prim '- (list (Prim 'read '()))))
  (Prim '+ (list (Var 'x) (Int 42)))))
```

3. $\boxed{\text{12 points}}$ The following is a partial impelmentation of the type checker for expressions of the $\mathcal{L}_{\mathsf{if}}^{mon}$ language, which includes integers, Booleans, conditionals, and several primitive operations. The `env` parameter is a dictionary that maps every in-scope variable to a type. Fill in the blanks of this type checker.

```
(define (type-check-exp env)
  (lambda (e)
    (match e
      [(Bool b) (values (Bool b) 'Boolean)]
      [(Let x e body)
       (define-values (e^ Te) ((type-check-exp env) e))
       (define-values (b Tb) ((type-check-exp ___(a)___) body))
       (values (Let x e^ b) ___(b)___)]
      [(If cnd thn els)
       (define-values (cnd^ Tc) ((type-check-exp env) cnd))
       (define-values (thn^ Tt) ___(c)___)
       (define-values (els^ Te) ((type-check-exp env) els))
       (check-type-equal? Tc ___(d)___)
       (check-type-equal? Tt ___(e)___)
       (values (If cnd^ thn^ els^) ___(f)___)]
      ...)))
```

Solution: (2 points each)

```
(a) (dict-set env x Te)
(b) Tb
(c) ((type-check-exp env) thn)
(d) 'Boolean
(e) Te
(f) Te  (or Tt)
```

4. $\boxed{\text{12 points}}$ Fill in the blanks to complete the cases for `If` in the following implementation of `rco-exp` and `rco-atom` (Remove Complex Operands), which translate from the $\mathcal{L}_{\mathsf{If}}$ language into $\mathcal{L}_{\mathsf{if}}^{mon}$. The grammars for these languages can be found in the Appendix of this exam. Recall that `rco-atom` must produce an atomic expression and an association list of variables and expressions. `rco-exp` returns an expression (which does not have to be atomic).

```
(define (rco-atom e)
  (match e
    [(Let x rhs body)
     (define new-rhs (rco-exp rhs))
     (define-values (new-body body-ss) (rco-atom body))
     (values new-body (append ___(a)___ body-ss))]
    [(Bool b) (values (Bool b) '())]
    [(If cnd thn els)
     (define if-exp (If ___(b)___ (rco-exp thn) (rco-exp els)))
     (define tmp (gensym 'tmp))
     (values ___(c)___ `((,tmp . ,___(d)___)))]
    ...))

(define (rco-exp e)
  (match e
    [(Let x rhs body)
     (Let x (rco-exp rhs) (rco-exp body))]
    [(Bool b) (Bool b)]
    [(If cnd thn els)
     (define cnd^ ___(e)___)
     (define thn^ (rco-exp thn))
     (define els^ (rco-exp els))
     ___(f)___]
    ...))
```

Solution: (2 points each)

```
(a) `((,x . ,new-rhs))
(b) (rco-exp cnd)
(c) (Var tmp)
(d) if-exp
(e) (rco-exp cnd)
(f) (If cnd^ thn^ els^)
```

5. $\boxed{\text{10 points}}$ Translate the following $\mathcal{L}_{\mathsf{if}}^{mon}$ program into $\mathcal{C}_{\mathsf{If}}$. The grammar for $\mathcal{C}_{\mathsf{If}}$ is in the Appendix of this exam.

```
(if (let ([tmp7 (read)])
       (eq? tmp7 0))
    (let ([tmp8 (read)])
       (- tmp8))
    (read))
```

Solution: (Approx. 1 point per statement.)

```
start:
    tmp7 = (read);
    if (eq? tmp7 0)
       goto block9;
    else
       goto block0;
block0:
    return (read);
block9:
    tmp8 = (read);
    return (- tmp8);
```

6. 14 points Given the following psuedo-x86 program, compile it to an equivalent and complete x86 program, using stack locations (not registers) for the variables. Your answer should be given in the AT&T syntax that the GNU assembler expects for .s files.

```
start:
    callq read_int
    movq %rax, x
    movq $-4, t0
    movq t0, t1
    addq x, t1
    movq t1, %rdi
    callq print_int
    movq $0, %rax
    jmp conclusion
```

Solution:

```
        .globl main
main:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    jmp start
start:
    callq read_int
    movq %rax, -16(%rbp)
    movq $-4, -8(%rbp)
    movq -16(%rbp), %rax
    addq %rax, -8(%rbp)
    movq -8(%rbp), %rdi
    callq print_int
    movq $0, %rax
    jmp conclusion
conclusion:
    addq $32, %rsp
    popq %rbp
    retq
```

Rubric:

- prelude (3 points)
- correct use of the stack for the variables (2 points)
- call to `read_int` and move from `rax` (2 point)
- at most one memory argument per instruction (2 points)
- move to `rdi` and call to `print_int` (2 points)
- conclusion (3 points)

7. 10 points Apply liveness analysis to the following pseudo-x86 program to determine the set of live locations before and after every instruction. (The callee and caller saved registers are listed in the Appendix of this exam.)

```
start:
    movq $0, sum

    movq $5, i

    jmp block.0


block.0:
    cmpq $0, i

    jg block.2

    jmp block.3


block.3:
    jmp block.1

block.2:
    addq i, sum

    subq $1, i

    jmp block.0


block.1:
    movq sum, %rdi

    callq print_int

    movq $0, %rax

    jmp conclusion
```

Solution:

```
block.1:
                {sum}
  movq sum, %rdi
                {%rdi}
  callq print_int
                {}
  movq $0, %rax
                {%rax}
  jmp conclusion
                {%rax}
block.2:
                {sum, i}
  addq i, sum
                {sum, i}
  subq $1, i
                {sum, i}
  jmp block.0
                {sum, i}

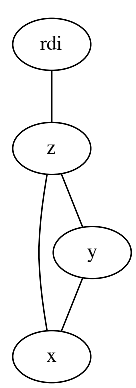
block.3:
                {sum}
  jmp block.1
                {sum}
block.0:
                {sum, i}
  cmpq $0, i
                {sum, i}
  jg block.2
                {sum, i}
  jmp block.3
                {sum, i}
start:
                {}
  movq $0, sum
                {sum}
  movq $5, i
                {sum, i}
  jmp block.0
                {sum, i}
```

8. 10 points Given the following results from liveness analysis, draw the interference graph. (The callee and caller saved registers are listed in the Appendix of this exam.)

```
start:
                {}
  callq _read_int
                {%rax}
  movq %rax, x
                {x}
  movq x, y
                {y, x}
  addq $1, y
                {y, x}
  movq y, z
                {y, x, z}
  addq $1, z
                {y, z, x}
  cmpq $0, x
                {y, x, z}
  je block.1
                {y, x, z}
  jmp block.2
                {y, z, x}
```

```
block.1:
                {x, z}
  movq x, %rdi
                {%rdi, z}
  callq print_int
                {z}
  jmp block.0
                {z}
block.2:
                {y, z}
  movq y, %rdi
                {%rdi, z}
  callq print_int
                {z}
  jmp block.0
                {z}
block.0:
                {z}
  movq z, %rdi
                {%rdi}
  callq print_int
                {}
  movq $0, %rax
                {%rax}
  jmp conclusion
                {%rax}
```

Solution:



9. 10 points Fill in the blanks to complete the following graph coloring algorithm.

```
(define (make-pqueue <=? [init '()]) ...)
(define (pqueue-push! q key) ...)
(define (pqueue-pop! q) ...)
(define (pqueue-decrease-key! q node) ...)
(define (pqueue-count q) ...)

(define (color-graph interfere-graph move-graph info)
  (define locals (dict-keys (dict-ref info 'locals-types)))
  (define unavailable-colors (make-hash))
  (define (compare u v)
    (>= (set-count (hash-ref unavailable-colors u))
        (set-count (hash-ref unavailable-colors v))))
  (define Q (make-pqueue ___(a)___))
  (define pq-node (make-hash))
  (define color (make-hash))
  (for ([r registers-for-alloc])
    (hash-set! color r (register->color r)))
  (for ([x locals])
       (define adj-reg
          (filter (lambda (u) (set-member? registers u))
                  (get-neighbors interfere-graph x)))
       (define adj-colors (list->set (map register->color adj-reg)))
       (hash-set! unavailable-colors x adj-colors)
       (hash-set! pq-node x ___(b)___))
  (while ___(c)___
         (define v (pqueue-pop! Q))
         (define move-related
           (sort (filter (lambda (x) (>= x 0))
                         (map (lambda (k) (hash-ref color k -1))
                              (get-neighbors move-graph v)))
                 <))
         (define c (choose-color v (hash-ref unavailable-colors v)
                                 move-related info))
         (hash-set! color v c)
         (for ([u ___(d)___])
              (when (not (set-member? registers u))
                    (hash-set! unavailable-colors u ___(e)___)
                    (pqueue-decrease-key! Q (hash-ref pq-node u)))))
  color)
```

Solution: (2 points each)

```
(a) compare
(b) (pqueue-push! Q x)
(c) (> (pqueue-count Q) 0)
(d) (in-neighbors interfere-graph v)
(e) (set-add (hash-ref unavailable-colors u) c)
```

Appendix

The caller-saved registers are:

```
rax rcx rdx rsi rdi r8 r9 r10 r11
```

and the callee-saved registers are:

```
rsp rbp rbx r12 r13 r14 r15
```

Grammar for $\mathcal{L}_{\mathsf{If}}$

type	::=	`Integer`
op	::=	`read` | `+` | `-`
exp	::=	`(Int` *int*`)` | `(Prim` *op* `(`*exp* ...`))`
exp	::=	`(Var` *var*`)` | `(Let` *var exp exp*`)`
type	::=	`Boolean`
bool	::=	`#t` | `#f`
cmp	::=	`eq?` | `<` | `<=` | `>` | `>=`
op	::=	*cmp* | `and` | `or` | `not`
exp	::=	`(Bool` *bool*`)` | `(If` *exp exp exp*`)`
$\mathcal{L}_{\mathsf{If}}$	::=	`(Program '()` *exp*`)`

Grammar for $\mathcal{L}_{\mathsf{if}}^{mon}$

atm	::=	`(Int` *int*`)` | `(Var` *var*`)`
exp	::=	*atm* | `(Prim 'read ())`
	|	`(Prim '- (`*atm*`))` | `(Prim '+ (`*atm atm*`))` | `(Prim '- (`*atm atm*`))`
	|	`(Let` *var exp exp*`)`
atm	::=	`(Bool` *bool*`)`
exp	::=	`(Prim not (`*atm*`))` | `(Prim` *cmp* `(`*atm atm*`))` | `(If` *exp exp exp*`)`
$\mathcal{L}_{\mathsf{if}}^{mon}$	::=	`(Program ()` *exp*`)`

Grammar for $\mathcal{C}_{\mathsf{If}}$

atm	::=	`(Int` *int*`)` | `(Var` *var*`)`
exp	::=	*atm* | `(Prim 'read ())` | `(Prim '- (`*atm*`))`
	|	`(Prim '+ (`*atm atm*`))` | `(Prim '- (`*atm atm*`))`
stmt	::=	`(Assign (Var` *var*`)` *exp*`)`
tail	::=	`(Return` *exp*`)` | `(Seq` *stmt tail*`)`
atm	::=	`(Bool` *bool*`)`
cmp	::=	`eq?` | `<` | `<=` | `>` | `>=`
exp	::=	`(Prim 'not (`*atm*`))` | `(Prim '`*cmp* `(`*atm atm*`))`
tail	::=	`(Goto` *label*`)`
	|	`(IfStmt (Prim` *cmp* `(`*atm atm*`)) (Goto` *label*`) (Goto` *label*`))`
$\mathcal{C}_{\mathsf{If}}$	::=	`(CProgram` *info* `((`*label* `.` *tail*`)` ...`))`