Compilers (Racket)
CSCI P423/523, Fall 2022

Final

Name: ____________________

This exam has 12 questions, for a total of 100 points.

1. [5 points] What is the observable behavior of the following $\mathcal{L}_{\mathsf{Fun}}$ program? (e.g. does it produce an error at compile time or runtime? does it produce an integer, which one? does it diverge?)

```
(define (f [x : Integer]) : (Vector Integer)
  (let ([v (vector (+ x x))])
    v))

(let ([v1 (f 3)])
  (let ([v2 (f 3)])
    (let ([v3 v1])
      (if (eq? v1 v2)
          (if (eq? v1 v3) 0 1)
          (if (eq? v1 v3) 2 3)))))
```

Solution: This program produces the number `2` because `(eq? v1 v2)` is false (they are vectors of different identity) and `(eq? v1 v3)` is true (they are aliases for the same vector).

2. [5 points] What is the observable behavior of the following $\mathcal{L}_{\mathsf{Tup}}$ program? (e.g. does it produce an error at compile time or runtime? does it produce an integer, which one? does it diverge?)

```
(let ([v1 (vector 3)])
  (let ([v2 (vector 3)])
    (if (eq? (vector-ref v1 0)
             (vector-ref v2 1))
        0
        1)))
```

Solution: The program is not well-typed because the type of `v2` is `(Vector Integer)` but `(vector-ref v2 1)` tries to access its second element. So the type checker outputs and error.

3. 8 points Given the following $\mathcal{L}_{\mathsf{While}}$ program, apply the Explicate Control pass to translate it to $\mathcal{C}_{\circlearrowleft}$. (You may assume that the following program was the result of the previous pass, Remove Complex Operands, which removes the `get!`s introduced by the pass Uncover `get!`.)

```
(let ([sum7 0])
  (let ([i8 (read)])
    (begin
      (while (> i8 0)
        (begin
          (set! sum7 (+ sum7 i8))
          (set! i8 (- i8 1))))
      (+ 27 sum7))))
```

Solution: (2 points per basic block)

```
start:
    sum7 = 0;
    i8 = (read);
    goto loop4;
loop4:
    if (> i8 0)
       goto block6;
    else
       goto block5;
block6:
    sum7 = (+ sum7 i8);
    i8 = (- i8 1);
    goto loop4;
block5:
    return (+ 27 sum7);
```

4. [13 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:
    callq read_int
    movq %rax, x57
    movq $1, y58
    callq read_int
    movq %rax, i59
    jmp loop65

loop65:
    movq i59, tmp60
    cmpq $0, tmp60
    jg block67
    jmp block66

block66:
    movq y58, tmp64
    movq x57, %rax
    addq tmp64, %rax
    jmp conclusion

block67:
    movq y58, tmp61
    movq y58, tmp62
    movq tmp61, y58
    addq tmp62, y58
    movq i59, tmp63
    movq tmp63, i59
    subq $1, i59
    jmp loop65
```

Solution: (1/2 point per liveness set)

```
start:
   (set)
callq read_int
   (set)
movq %rax, x57
      { x57}
movq $1, y58
      { x57 y58}
callq read_int
      { x57 y58}
movq %rax, i59
      { x57 y58 i59}
jmp loop65
      { x57 y58 i59}

loop65:
      { x57 y58 i59}
movq i59, tmp60
      { x57 y58 i59 tmp60}
cmpq $0, tmp60
      { x57 y58 i59}
jg block67
      { x57 y58 i59}
jmp block66
      { x57 y58}

block66:
      { x57 y58}
movq y58, tmp64
      { tmp64 x57}
movq x57, %rax
      { tmp64}
addq tmp64, %rax
      {}
jmp conclusion
      {}

block67:
      { x57 y58 i59}
movq y58, tmp61
      { tmp61 x57 y58 i59}
movq y58, tmp62
      { tmp61 tmp62 x57 i59}
movq tmp61, y58
      { tmp62 x57 y58 i59}
addq tmp62, y58
      { x57 y58 i59}
movq i59, tmp63
      { tmp63 x57 y58}
movq tmp63, i59
      { x57 y58 i59}
subq $1, i59
      { x57 y58 i59}
jmp loop65
      { x57 y58 i59}
```

5. 10 points Fill in the blanks for the following `expose-alloc-vector` auxilliary function of the Expose Allocation pass that translates from $\mathcal{L}_{\mathsf{Tup}}$ to $\mathcal{L}_{\mathsf{Alloc}}$. (The grammar for $\mathcal{L}_{\mathsf{Alloc}}$ is in the Appendix.)

```
(define/public (expose-alloc-vector e* vec-type alloc-exp)
  (define vec (gensym 'alloc))
  (define-values (bindingss inits)
    (for/lists (l1 l2) ([e e*])
      (cond [(atm? e) (values '() e)]
            [else
             (define tmp (gensym 'vecinit))
             (values (list (cons tmp e)) (Var tmp))])))
  (define bindings (append* bindingss))
  (define initialize-vec
    (foldr
     (lambda (init n rest)
       (let ([v (gensym '_)])
         (Let v
              ___(a)___
              rest)))
     (Var vec) inits (range (length e*))))
  (define voidy (gensym '_))
  (define num-bytes ___(b)___)
  (define alloc-init-vec
    (Let voidy
      (If (Prim '< (list ___(c)___
                    (GlobalValue 'fromspace_end)))
          (Void)
          ___(d)___)
      (Let vec alloc-exp initialize-vec)))
  (make-lets bindings alloc-init-vec))

(define/public (expose-alloc-exp e)
  (match e
    [(HasType (Prim 'vector es) vec-type)
     (expose-alloc-vector
       ___(e)___
       vec-type
       (Allocate (length es) vec-type))]
    ...))
```

Solution: (2 points each)

```
(a) (Prim 'vector-set! (list (Var vec) (Int n) init))
(b) (* (+ (length e*) 1) 8)
(c) (Prim '+ (list (GlobalValue 'free_ptr) (Int num-bytes)))
(d) (Collect num-bytes)
(e) (for/list ([e es]) (expose-alloc-exp e))
```

6. [4 points] In the `expose-alloc-vector` function of the previous question, why are the intializing expressions `e*` bound with `let` expressions (with the `make-lets` at the bottom) instead of using them directly in the vector initialization?

> **Solution:** The reason is that between the allocation of the vector and the initialization of its elements, we cannot allow a call to `collect` because then the garbage collector would try to traverse a partially-initialized vector, causing it to potentially jump to random locations in memory. The initializing expressions may contain vector-creation expressions and hence calls to `collect`. The `let` binding of the intializing expressions causes them to be executed first, prior to the allocation and initialization of the vector.

7. [6 points] Describe the layout of the 64-bit tag at the beginning of every tuple.

> **Solution:**
>
> - Bit position 0 is set to 0 when the tuple has been copied into the TO space (in the process of doing a copy collection) and it is set to 1 otherwise. If it is set to 0, then the 64 bits are the address of the new location in the TO space. **(2 points)**
> - Bit position 1 through 6 stores the length of the tuple. **(2 points)**
> - Bit position 7 through 57 is the pointer mask. It says, for each element of the tuple, whether the element is a pointer (that is, a tuple) or something else (like an integer or Boolean). **(2 points)**

8. 10 points Fill in the blanks for the following `explicate-control` that translates $\mathcal{L}^{mon}_{\mathsf{FunRef}}$ programs into $\mathcal{C}_{\mathsf{Fun}}$ programs.

```
(define/override (explicate-assign e x cont-block)
  (match e
    [(Apply f arg*) ___(a)___]
    ...))
(define/override (explicate-tail e)
  (match e
    [(Apply f arg*) ___(b)___]
    ...))
(define/override (explicate-pred cnd thn-block els-block)
  (match cnd
    [(Apply f arg*)
     (define tmp (gensym 'tmp))
     (Seq ___(c)___
          (IfStmt (Prim 'eq? (list (Var tmp) (Bool #t)))
                  (create_block thn-block)
                  (create_block els-block)))]
    ...))
(define/override (explicate-effect e cont-block)
  (match e
    [(Apply f arg*) ___(d)___]
    ...))
(define/public (explicate-control-def d)
  (match d
    [(Def f params ty info body)
     (set! basic-blocks '())
     (define body-block ___(e)___)
     (define new-blocks (dict-set basic-blocks
          (symbol-append f 'start) body-block))
     (Def f params ty info new-blocks)]
    ))
```

Solution: (2 points each)

```
(a) (Seq (Assign (Var x) (Call f arg*)) cont-block)
(b) (TailCall f arg*)
(c) (Assign (Var tmp) (Call f arg*)
(d) (Seq (Call f arg*) cont-block)
(e) (explicate-tail body)
```

9. 6 points What is the purpose of the Reveal Functions pass? How does the output of Reveal Functions facilitate decisions made in later passes of the compiler?

Solution: The Reveal Functions pass separates the references to global functions from references to local variables, changing the former to `FunRef`. **(2 points)** The later passes can then treat them differently, in particular,

- In Remove Complex Operands, `FunRef` is treated as a complex operand to make sure it only appears on the righ-hand side of an assignment statement. **(2 points)**
- In Instruction Selection, each assignment with `FunRef` on the right-hand side is translated to a `leaq` instruction that obtains the function's address from the function's label. **(2 points)**

10. 8 points Describe the general layout of the procedure call frame that your compiler uses.

Solution: The procedure call frame stores the following information:

- The return address, i.e., the address of the caller. **(2 points)**
- The caller's value for `rbp`. **(2 points)**
- The caller's values of the callee-saved registers that are going to be used in this function. **(2 points)**
- The spilled local variables. **(2 points)**

11. 13 points Apply Instruction Selection to the following two functions, translating them from $\mathcal{C}_{\mathsf{Fun}}$ to $\text{x86}^{\mathsf{Def}}_{\mathsf{callq*}}$. (The definitions of $\mathcal{C}_{\mathsf{Fun}}$ and $\text{x86}^{\mathsf{Def}}_{\mathsf{callq*}}$ are in the Appendix, as is the list of argument-passing registers.) (The function `even_57` calls `odd_58`, but you do not need to translate the `odd_58` function for this exam question, so its definition is omitted.)

```
(define (even_57 [x59 : Integer]) : Boolean
   even_57start:
      if (eq? x59 0)
         goto block69;
      else
         goto block70;
   block70:
      tmp61 = (fun-ref odd_58 1);
      tmp62 = (- 1);
      tmp63 = (+ tmp62 x59);
      (tail-call tmp61 tmp63)
   block69:
      return #t;
)

(define (main) : Integer
   mainstart:
      tmp67 = (fun-ref even_57 1);
      tmp68 = (read);
      tmp73 = (call tmp67 tmp68);
      if (eq? tmp73 #t)
         goto block74;
      else
         goto block75;
   block75:
      return 42;
   block74:
      return 999;
)
```

Solution: (approx. 1/2 point per instruction)

```
(define (even_57) : Integer
   block70:
      leaq odd_58(%rip), tmp61
      movq $1, tmp62
      negq tmp62
      movq tmp62, tmp63
      addq x59, tmp63
      movq tmp63, %rdi
      tail-jmp tmp61

   block69:
      movq $1, %rax
      jmp even_57conclusion

   even_57start:
      movq %rdi, x59
```

```
        cmpq $0, x59
        je block69
        jmp block70
)

(define (main) : Integer
    block75:
        movq $42, %rax
        jmp mainconclusion

    block74:
        movq $999, %rax
        jmp mainconclusion

    mainstart:
        leaq even_57(%rip), tmp67
        callq read_int
        movq %rax, tmp68
        movq tmp68, %rdi
        callq *tmp67
        movq %rax, tmp73
        cmpq $1, tmp73
        je block74
        jmp block75
)
```

12. [12 points] Draw the interference graph for the following program fragment by adding edges between the nodes below. You do not need to include edges between two registers. The live-after set for each instruction is given to the right of each instruction and the types of each variable is listed below. (The callee and caller saved registers are listed in the Appendix of this exam.)

```
(define (main) : Integer
  locals-types:
    tmp73 : Integer, tmp72 : Integer, tmp75 : (Integer -> Integer),
    tmp74 : ((Integer -> Integer) (Vector Integer Integer) -> Void), tmp71 : Integer,
    _65 : Void, _64 : Void, _66 : Void, alloc63 : (Vector Integer Integer),
    vec62 : (Vector Integer Integer)
```

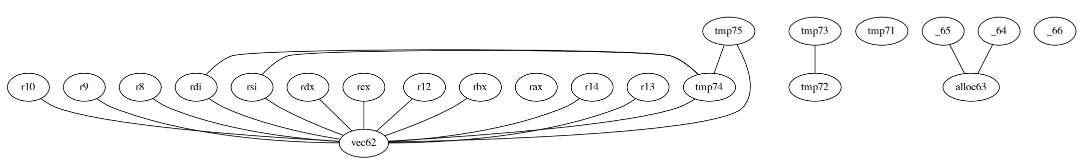
```
mainstart:
    {}
movq free_ptr(%rip), tmp71
    {tmp71}
movq tmp71, tmp72
    {tmp72}
addq $24, tmp72
    {tmp72}
movq fromspace_end(%rip), tmp73
    {tmp72 tmp73}
cmpq tmp73, tmp72
    {}
jl block77
    {}
jmp block78
    {}

block77:
     {}
movq $0, _66
    {}
jmp block76
    {}

block78:
    {}
movq %r15, %rdi
    {rdi}
movq $24, %rsi
    {rdi rsi}
callq collect
    {}
jmp block76
    {}
```

```
block76:
    {}
movq free_ptr(%rip), %r11
    {}
addq $24, free_ptr(%rip)
    {}
movq $5, 0(%r11)
    {r11}
movq %r11, alloc63
    {alloc63}
movq alloc63, %r11
    {alloc63}
movq $0, 8(%r11)
    {alloc63}
movq $0, _65
    {alloc63}
movq alloc63, %r11
    {alloc63}
movq $41, 16(%r11)
    {alloc63}
movq $0, _64
    {alloc63}
movq alloc63, vec62
    {vec62}
leaq map_vec_57(%rip), tmp74
    {vec62 tmp74}
leaq add158(%rip), tmp75
    {vec62 tmp74 tmp75}
movq tmp75, %rdi
    {tmp74 rdi vec62}
movq vec62, %rsi
    {rsi tmp74 rdi vec62}
callq *tmp74
    {vec62}
movq vec62, %r11
    {r11}
movq 16(%r11), %rax
    {rax}
jmp mainconclusion
    {rax}
)
```

Solution:



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
```

The argument-passing registers are:

```
rdi rsi rdx rcx r8 r9
```

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

```
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)
-----------------------------------------------------------------
type   ::=  Void
exp    ::=  (SetBang var exp) | (Begin exp* exp) | (WhileLoop exp exp) | (Void)
L_While ::=  (Program '() exp)
```

Grammar for $\mathcal{C}_{\circlearrowleft}$

```
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))
-----------------------------------------------------------------
atm   ::=  (Void)
stmt  ::=  (Prim 'read ())
C_↺   ::=  (CProgram info ((label . tail) ...))
```

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

```
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)
---------------------------------------------------------------
type  ::= Void
exp   ::= (SetBang var exp) | (Begin exp* exp) | (WhileLoop exp exp) | (Void)
---------------------------------------------------------------
type  ::= (Vector type*)
op    ::= vector | vector-length
exp   ::= (Prim vector-ref (exp (Int int)))
       |  (Prim vector-set! (exp (Int int) exp))
L_Tup ::= (Program '() exp)
```

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

The $\mathcal{L}_{\mathsf{Alloc}}$ language extends $\mathcal{L}_{\mathsf{Tup}}$ with the following:

```
exp ::= (Collect int) | (Allocate int type) | (GlobalValue name)
```

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

```
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)
---------------------------------------------------------------
type  ::= Void
exp   ::= (SetBang var exp) | (Begin exp* exp) | (WhileLoop exp exp) | (Void)
---------------------------------------------------------------
type  ::= (Vector type*)
op    ::= vector | vector-length
exp   ::= (Prim vector-ref (exp (Int int)))
       |  (Prim vector-set! (exp (Int int) exp))
---------------------------------------------------------------
type  ::= (type ... -> type)
exp   ::= (Apply exp exp ...)
def   ::= (Def var ([var:type] ...) type '() exp)
L_Fun ::= (ProgramDefsExp '() (def ...)) exp)
```

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

```
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)
--------------------------------------------------------------------------
atm   ::=  (Void)
exp   ::=  (GetBang var) | (SetBang var exp) | (Begin (exp ...) exp)
       |   (WhileLoop exp exp)
--------------------------------------------------------------------------
exp   ::=  (Collect int)) | (Allocate int type) | (GlobalValue var)
--------------------------------------------------------------------------
type  ::=  (type ... -> type)
exp   ::=  (FunRef label int) | (Apply atm atm ...)
def   ::=  (Def var ([var:type] ...) type '() exp)
L^mon_FunRef  ::=  (ProgramDefsExp '() (def ...)) exp)
```

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

```
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))
--------------------------------------------------------------------------
atm   ::=  (Void)
stmt  ::=  (Prim 'read ())
--------------------------------------------------------------------------
exp   ::=  (Allocate int type)
       |   (Prim vector-ref (atm (Int int)))
       |   (Prim vector-set! (atm (Int int) atm))
       |   (Prim vector-length (atm))
       |   (GlobalValue var)
stmt  ::=  (Prim vector-set! (atm (Int int) atm))
       |   (Collect int)
--------------------------------------------------------------------------
exp   ::=  (FunRef label int) | (Call atm (atm ...))
tail  ::=  (TailCall atm atm ...)
def   ::=  (Def label ([var:type] ...) type info ((label . tail) ...))
C_Fun ::=  (ProgramDefs info (def ...))
```

Grammar for $\text{x86}^{\mathsf{Def}}_{\mathsf{callq}*}$

```
reg    ::= rsp | rbp | rax | rbx | rcx | rdx | rsi | rdi |
           r8 | r9 | r10 | r11 | r12 | r13 | r14 | r15
arg    ::= (Imm int) | (Reg reg) | (Deref reg int)
instr  ::= (Instr addq (arg arg)) | (Instr subq (arg arg))
        |  (Instr negq (arg)) | (Instr movq (arg arg))
        |  (Instr pushq (arg)) | (Instr popq (arg))
        |  (Callq label int) | (Retq) | (Jmp label)
block  ::= (Block info (instr ...))
-------------------------------------------------------------
bytereg ::= ah | al | bh | bl | ch | cl | dh | dl
arg     ::= (ByteReg bytereg)
cc      ::= e | l | le | g | ge
instr   ::= (Instr xorq (arg arg)) | (Instr cmpq (arg arg))
         |  (Instr set (cc arg)) | (Instr movzbq (arg arg))
         |  (JmpIf cc label)
-------------------------------------------------------------
arg  ::= (Global label)
-------------------------------------------------------------
instr  ::= (IndirectCallq arg int) | (TailJmp arg int)
        |  (Instr 'leaq (arg (Reg reg)))
block  ::= (Block info (instr ...))
def    ::= (Def label '() type info ((label . block) ...))
x86Def_callq* ::= (X86Program info (def ...))
```