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

1. **5 points** What is the observable behavior of the following $L_{\text{Fun}}$ 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 $L_{\text{Tup}}$ 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 $L_{\text{While}}$ program, apply the Explicate Control pass to translate it to $C_{\mathbb{C}}$. (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!.)

$$
\begin{align*}
(\text{let } ([\text{sum7 } 0])
\quad (\text{let } ([\text{i8 } (\text{read})])
\quad \text{(begin}
\quad \quad (\text{while } (> \text{i8 } 0))
\quad \quad \text{(begin}
\quad \quad \quad (\text{set! } \text{sum7 } (+ \text{sum7 } \text{i8}))
\quad \quad \quad (\text{set! } \text{i8 } (- \text{i8 } 1)))
\quad \quad (+ 27 \text{sum7})))
\end{align*}
$$

**Solution**: (2 points per basic block)

```
start:  
  \text{sum7} = 0;
  \text{i8} = (\text{read});
  \text{goto } \text{loop4};

\text{loop4}: 
  \text{if } (> \text{i8 } 0) 
  \text{goto } \text{block6};
  \text{else}
  \text{goto } \text{block5};

\text{block6}: 
  \text{sum7} = (+ \text{sum7 } \text{i8});
  \text{i8} = (- \text{i8 } 1);
  \text{goto } \text{loop4};

\text{block5}: 
  \text{return } (+ 27 \text{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:

block67:
```

**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
  { tmp64}
jmp conclusion
  { tmp64}

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 $\mathcal{L}_{\text{Tup}}$ to $\mathcal{L}_{\text{Alloc}}$. (The grammar for $\mathcal{L}_{\text{Alloc}}$ 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 (c) 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 initializing 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 initializing 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 $L^{\text{mon}}_{\text{FunRef}}$ programs into $\mathcal{C}_{\text{Fun}}$ programs.

```scheme
(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*)) (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 \texttt{FunRef}. (2 points) The later passes can then treat them differently, in particular,

- In Remove Complex Operands, \texttt{FunRef} is treated as a complex operand to make sure it only appears on the right-hand side of an assignment statement. (2 points)
- In Instruction Selection, each assignment with \texttt{FunRef} on the right-hand side is translated to a \texttt{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 \texttt{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 \texttt{CFun} to \texttt{x86Def}. (The definitions of \texttt{CFun} and \texttt{x86Def} are in the Appendix, as is the list of argument-passing registers.) (The function \texttt{even\_57} calls \texttt{odd\_58}, but you do not need to translate the \texttt{odd\_58} function for this exam question, so its definition is omitted.)

\begin{verbatim}
(define (even\_57 \[x59 : \text{Integer}\]) : \text{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) : \text{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;
)
\end{verbatim}

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

\begin{verbatim}
(define (even\_57) : \text{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
\end{verbatim}
```scheme
(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  
    {}  
  movq tmp71, tmp72  
    {}  
  addq $24, tmp72  
    {}  
  movq fromspace_end(%rip), tmp73  
    {}  
  cmpq tmp73, tmp72  
    {}  
  jl block77  
    {}  
  jmp block78  
    {}  
  block77:
    {}  
  movq $0, _66  
    {}  
  jmp block76  
    {}  
  block76:
    {}  
  movq %r15, %rdi  
    {}  
  movq $24, %rsi  
    {}  
  callq collect  
    {}  
  jmp block76  
    {}  
  block78:
    {}  
  movq %r15, %rdi  
    {}  
  movq $24, %rsi  
    {}  
  callq *tmp74  
    {}  
  jmp mainconclusion  
    {}

Solution:
```

![Interference Graph Image]
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 $L_{\text{While}}$

\[
\begin{align*}
type & ::= \text{Integer} \\
op & ::= \text{read} | + | - \\
exp & ::= (\text{Int int}) | (\text{Prim op exp} \ldots) \\
\end{align*}
\]

\[
\begin{align*}
exp & ::= (Var var) | (Let var exp exp) \\
\end{align*}
\]

Grammar for $C$
Grammar for $\mathcal{L}_{\text{Tup}}$


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)  

L_{\text{Tup}} ::= (Program '() exp)

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

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

\[
exp ::= (\text{Collect int}) | (\text{Allocate int type}) | (\text{GlobalValue name})
\]

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

\[
type ::= Integer
\]

\[
exp ::= (Int int) | (Prim op exp...)
\]

\[
exp ::= (Var var) | (Let var exp exp)
\]

\[
exp ::= (SetBang var exp) | (Begin exp* exp) | (WhileLoop exp exp) | (Void)
\]

\[
type ::= (Vector type*)
\]

\[
exp ::= (Prim vector-ref (exp (Int int)))
\]

\[
exp ::= (Prim vector-set! (exp (Int int) exp))
\]

\[
\mathcal{L}_{\text{Fun}} ::= (ProgramDefsExp '() (def...)) exp
\]
Grammar for $L_{\text{FunRef}}^{\text{mon}}$

$\text{atm} ::= (\text{Int } \text{int}) \mid (\text{Var } \text{var})$
$\text{exp} ::= \text{atm} \mid (\text{Prim 'read ()})$
$\text{exp} ::= \text{atm} \mid (\text{Prim '}(\text{atm atm})) \mid (\text{Prim '-}(\text{atm atm}))$
$\text{exp} ::= \text{Let var exp exp}$

$\text{atm} ::= (\text{Bool } \text{bool})$
$\text{exp} ::= (\text{Prim 'not (atm)}) \mid (\text{Prim cmp (atm atm)}) \mid (\text{If exp exp exp})$
$\text{exp} ::= (\text{GetBang var}) \mid (\text{SetBang var exp}) \mid (\text{Begin (exp... exp)})$
$\text{exp} ::= (\text{WhileLoop exp exp})$
$\text{exp} ::= (\text{Collect int})) \mid (\text{Allocate int type}) \mid (\text{GlobalValue var})$
$\text{type} ::= (\text{type... -> type})$
$\text{exp} ::= (\text{FunRef label int}) \mid (\text{Apply atm atm...})$
$\text{def} ::= (\text{Def var [var:type]... type '() exp})$

$L_{\text{FunRef}}^{\text{mon}} ::= (\text{ProgramDefsExp '() (def ...)) exp})$

Grammar for $C_{\text{Fun}}$

$\text{atm} ::= (\text{Int int}) \mid (\text{Var var})$
$\text{exp} ::= \text{atm} \mid (\text{Prim 'read ()}) \mid (\text{Prim '-}(\text{atm atm}))$
$\text{exp} ::= \text{atm} \mid (\text{Prim '+}(\text{atm atm})) \mid (\text{Prim '-}(\text{atm atm}))$
$\text{stmt} ::= (\text{Assign (Var var) exp})$
$\text{tail} ::= (\text{Return exp}) \mid (\text{Seq stmt tail})$

$\text{atm} ::= (\text{Bool bool})$
$\text{cmp} ::= \text{eq?} \mid < \mid \leq \mid > \mid \geq$
$\text{exp} ::= (\text{Prim 'not (atm)}) \mid (\text{Prim cmp (atm atm)})$
$\text{tail} ::= (\text{Goto label})$
$\text{tail} ::= (\text{IfStmt (Prim cmp (atm atm)) (Goto label) (Goto label)})$
$\text{atm} ::= (\text{Void})$
$\text{stmt} ::= (\text{Prim 'read ()})$

$\text{exp} ::= (\text{Allocate int type})$
$\text{exp} ::= (\text{Collect int}))$
$\text{exp} ::= (\text{FunRef label int}) \mid (\text{Call atm atm...})$
$\text{exp} ::= (\text{TailCall atm atm...})$
$\text{def} ::= (\text{Def label [var:type]... type info ((label . tail)...))}$

$L_{\text{FunRef}}^{\text{mon}} ::= (\text{ProgramDefs info (def ...))}$
Grammar for \( \text{x86}^{\text{Def}}_{\text{callq*}} \)

\[
\begin{align*}
\text{reg} & ::= \text{rsp} | \text{rbp} | \text{rax} | \text{rbx} | \text{rcx} | \text{rdx} | \text{rsi} | \text{rdi} | \\
& \quad | \text{r8} | \text{r9} | \text{r10} | \text{r11} | \text{r12} | \text{r13} | \text{r14} | \text{r15} \\
\text{arg} & ::= (\text{Imm} \text{ int}) | (\text{Reg} \text{ reg}) | (\text{Deref} \text{ reg} \text{ int}) \\
\text{instr} & ::= (\text{Instr} \text{ addq} (\text{arg} \text{ arg})) | (\text{Instr} \text{ subq} (\text{arg} \text{ arg})) \\
& \quad | (\text{Instr} \text{ negq} (\text{arg})) | (\text{Instr} \text{ movq} (\text{arg} \text{ arg})) \\
& \quad | (\text{Instr} \text{ pushq} (\text{arg})) | (\text{Instr} \text{ popq} (\text{arg})) \\
& \quad | (\text{Callq} \text{ label} \text{ int}) | (\text{Retq}) | (\text{Jmp} \text{ label}) \\
\text{block} & ::= (\text{Block} \text{ info} (\text{instr} \ldots)) \\
\text{bytereg} & ::= \text{ah} | \text{al} | \text{bh} | \text{bl} | \text{ch} | \text{cl} | \text{dh} | \text{dl} \\
\text{arg} & ::= (\text{ByteReg} \text{bytereg}) \\
\text{cc} & ::= \text{e} | \text{l} | \text{le} | \text{g} | \text{ge} \\
\text{instr} & ::= (\text{Instr} \text{ xorq} (\text{arg} \text{ arg})) | (\text{Instr} \text{ cmpq} (\text{arg} \text{ arg})) \\
& \quad | (\text{Instr} \text{ set} (\text{cc} \text{ arg})) | (\text{Instr} \text{ movzbq} (\text{arg} \text{ arg})) \\
& \quad | (\text{JmpIf} \text{ cc} \text{ label}) \\
\text{arg} & ::= (\text{Global} \text{ label}) \\
\text{instr} & ::= (\text{IndirectCallq} \text{ arg} \text{ int}) | (\text{TailJmp} \text{ arg} \text{ int}) \\
& \quad | (\text{Instr} \text{ 'leaq} (\text{arg} (\text{Reg} \text{ reg}))) \\
\text{block} & ::= (\text{Block} \text{ info} (\text{instr} \ldots)) \\
\text{def} & ::= (\text{Def} \text{ label} \text{ '()} \text{ type} \text{ info} (\text{label} \text{ block} \ldots)) \\
\text{x86}^{\text{Def}}_{\text{callq*}} & ::= (\text{X86Program} \text{ info} (\text{def} \ldots))
\end{align*}
\]