Compilers (Racket)
CSCI P423/523, Fall 2021

Final

Name: ____________________

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

1. [4 points] What is the output of the following Racket program?

```
(let ([a (vector (vector 0) 1)])
  (let ([b (vector-ref a 0)])
    (let ([c a])
      (begin
        (vector-set! c 0 (vector 1))
        (vector-ref b 0)))))
```

Solution:

```
0
```

2. [4 points] What is the output of the following Racket program?

```
(let ([a (vector (vector 0) 1)])
  (let ([b (vector-ref a 0)])
    (let ([c a])
      (begin
        (vector-set! (vector-ref c 0) 0 1)
        (vector-ref b 0)))))
```

Solution:

```
1
```

3. [4 points] What is the output of the following Racket program?

```
(define (f [x : Integer]) : Void
   (begin
      (set! x 0)
      (void)))

(let ([y 1])
   (begin
      (f y)
      y))
```

Solution:

```
1
```

4. $\boxed{\text{4 points}}$ Why does our compiler spill variables of `Vector` type to the root stack instead of the regular procedure call stack?

> **Solution:** We spill them to the root stack so that the garbage collector has easy access to all the live tuples. It separates them from the other non-vector variables that the garbage collector must ignore.

5. $\boxed{\text{4 points}}$ Why must the prelude of a function push the contents of the `rbp` register to the procedure call stack?

> **Solution:** The `rbp` register is a callee-saved register, so when we return from this function, its contents must be the same as they were upon entry to this function. But we change `rbp` in this function, so we have to restore its original value in the conclusion. Thus, we push it on the stack in the prelude and pop it back off in the conclusion.

6. 10 points Given the following input program to the Expose Allocation pass, what would be the output of Expose Allocation?

```
(let ([v3 (vector 42)])
   (vector-ref v3 0))
```

Solution: 2 points each

- Check for space
- Call to collect
- allocate
- initialize
- return the address

```
(let ([v3 (let ([_ (if (< (+ (global-value free_ptr) 16)
                          (global-value fromspace_end))
                       (void)
                       (collect 16))])
               (let ([alloc4 (allocate 1 (Vector Integer))])
                    (let ([_ (vector-set! alloc4 0 42)])
                         alloc4)))])
     (vector-ref v3 0))
```

7. 12 points Given the input program on the left, fill in the blanks in the output of Select Instructions on the right.

```
start:
    t8 = (global-value free_ptr);
    t9 = (+ t8 16);
    t0 = (global-value fromspace_end);
    if (< t9 t0)
       goto block2;
    else
    goto block3;


block2:
    t7 = (void);
    goto block1;


block3:
    (collect 16)
    goto block1;


block1:
    alloc5 = (allocate 1 (Vector Integer));
    t6 = (vector-set! alloc5 0 777);
    v3 = alloc5;
    t4 = (vector-set! v3 0 42);
    return (vector-ref v3 0);
```

```
start:
    movq ___(a)___, t8
    movq t8, t9
    addq $16, t9
    movq ___(b)___, t0
    cmpq t0, t9
    jl block2
    jmp block3

block2:
    movq $0, t7
    jmp block1

block3:
    movq %r15, %rdi
    movq $16, %rsi
    ___(c)___
    jmp block1

block1:
    movq free_ptr(%rip), %r11
    ___(d)___
    movq $3, 0(%r11)
    movq %r11, alloc5
    movq alloc5, %r11
    movq $777, 8(%r11)
    movq $0, t6
    movq alloc5, v3
    movq v3, %r11
    ___(e)___
    movq $0, t4
    movq v3, %r11
    ___(f)___
    jmp conclusion
```

Solution: (2 points each)

```
(a) free_ptr(%rip)
(b) fromspace_end(%rip)
(c) callq collect
(d) addq $16, free_ptr(%rip)
(e) movq $42, 8(%r11)
(f) movq 8(%r11), %rax
```

8. [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.

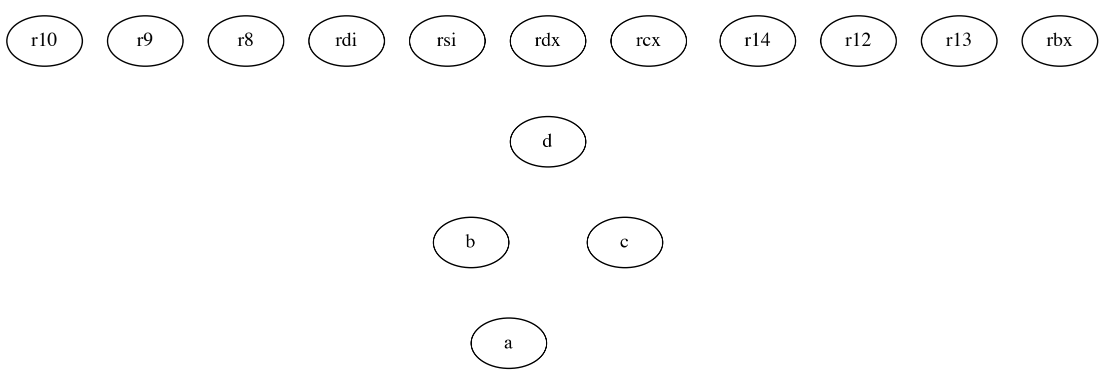
Recall that the caller-saved registers are

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

and the callee-saved registers are

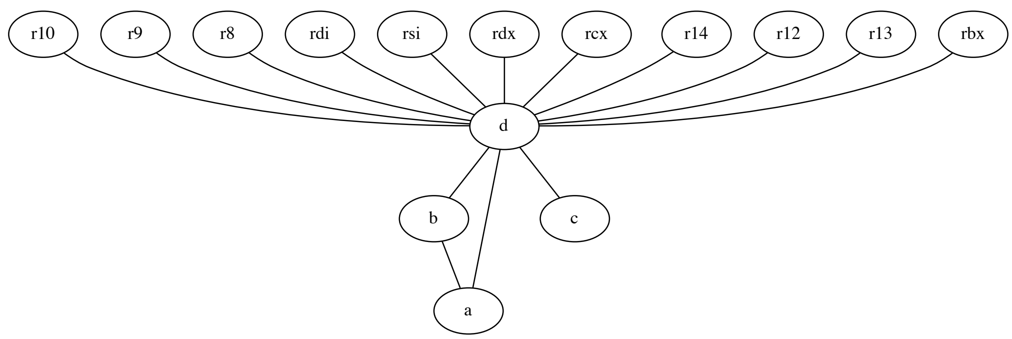
```
rsp rbp rbx r12 r13 r14 r15
```

```
a : Void, b : (Vector Integer), c : (Vector Integer), d : (Vector Integer)

block1:             { r15 d }
    movq %r15, %rdi  { rdi d }
    movq $16, %rsi   { rdi d rsi }
    callq collect    { d }
    jmp block2       { d }

block2:                       { d }
    movq free_ptr(%rip), %r11 { d }
    addq $16, free_ptr(%rip)  { d }
    movq $3, 0(%r11)          { r11 d }
    movq %r11, b              { b d }
    movq b, %r11              { b d }
    movq $0, 8(%r11)          { b d }
    movq $0, a                { b d }
    movq b, c                 { c d }
    cmpq c, d                 { }
    je block7                 { }
    jmp block8                { }
```



Solution:

- Edges between `d` and the caller-saved registers. (2 points)
- Edges between `d` and the callee-saved registers. (2 points)
- The edges between variables `a`, `b`, `c`, and `d`. (2 points each)



9. 12 points Given the following output of Remove Complex Operands, apply the Explicate Control pass to translate the program to $\mathcal{C}_{\mathsf{Fun}}$. You may use concrete or abstract syntax for your answer. Make sure to distinguish regular calls (`call` *fun* $arg_1 \ldots arg_n$) from tail calls (`tail-call` *fun* $arg_1 \ldots arg_n$).

```
(define (apply3 [f5 : (Integer -> Integer)] [x6 : Integer]) : Integer
   (let ([tmp8 (f5 x6)])
      (f5 tmp8)))

(define (inc4 [x7 : Integer]) : Integer
   (+ x7 1))

(define (main) : Integer
   (let ([tmp9 (fun-ref apply3)])
      (let ([tmp0 (fun-ref inc4)])
         (let ([tmp1 (read)])
            (tmp9 tmp0 tmp1)))))
```

Solution:

- Regular call inside `apply`. (2 points)
- Tail call inside `apply`. (2 points)
- Return statement inside `inc`. (2 point)
- Convert `let` to assignment statements. (2 points)
- Tail call inside `main`. (2 points)
- Start labels. (2 points)

```
(define (apply3 [f5 : (Integer -> Integer)] [x6 : Integer]) : Integer
   apply3start:
      tmp8 = (call f5 x6);
      (tail-call f5 tmp8))

(define (inc4 [x7 : Integer]) : Integer
   inc4start:
      return (+ x7 1);)

(define (main) : Integer
   mainstart:
      tmp9 = (fun-ref apply3);
      tmp0 = (fun-ref inc4);
      tmp1 = (read);
      (tail-call tmp9 tmp0 tmp1))
```

10. $\boxed{\text{12 points}}$ Given the following $\mathcal{C}_{\mathsf{Fun}}$ program, apply the Select Instructions pass.

```
(define (id3 [x4 : Integer]) : Integer
   id3start:
      return x4;)

(define (main) : Integer
   mainstart:
      tmp5 = (fun-ref id3);
      tmp6 = (call tmp5 41);
      return (+ 1 tmp6);
```

Recall that the following six registers are used for passing arguments to functions.

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

Solution:

```
(define (id3) : Integer    ;; no parameters      (1 point)
   id3start:
      movq %rdi, x4        ;; parameter passing (2 points)
      movq x4, %rax        ;; return x          (1 point)
      jmp id3conclusion)

(define (main) : Integer
   mainstart:
      leaq (fun-ref id3), tmp5 ;; FunRef            (2 points)
      movq $41, %rdi           ;; parameter passing (1 point)
      callq *tmp5              ;; indirect call      (2 points)
      movq %rax, tmp6          ;; call result        (1 point)
      movq $1, %rax            ;; + 1                (1 point)
      addq tmp6, %rax          ;; return             (1 point)
      jmp mainconclusion
```

11. 10 points Recall that the Limit Functions pass changes all the functions in the program so that they have at most 6 parameters (the number of argument-passing registers), making it easier to implement efficient tail calls. The `limit-type` auxiliary function changes each type annotation in the program as part of the Limit Functions pass. Fill in the blanks in `limit-type`.

```
(define (limit-type t)
  (match t
    [`(Vector ,ts ...)
      (define new-ts (for/list ([t ts]) ___(a)___))

      ___(b)___]

    [`(,ts ... -> ,rt)
     (define new-ts (for/list ([t ts]) (limit-type t)))
     (define new-rt (limit-type rt))
     (define n (vector-length arg-registers))
     (cond [(> (length new-ts) n)
            (define-values (first-ts last-ts) (split-at new-ts (- n 1)))
            ___(c)___]


           [else
            ___(d)___])]


    [else ___(e)___]



    ))
```

Solution: (2 points each)

```
(a) (limit-type t)
(b) `(Vector ,@new-ts)
(c) `(,@first-ts (Vector ,@last-ts) -> ,new-rt)
(d) `(,@new-ts -> ,new-rt)
(e) t
```

12. [12 points] Given the following x86 code for a function named `map_vec`, write down the code for its prelude and conclusion.

```
map_vecstart:
        movq    %rdi, -16(%rbp)
        movq    %rsi, -8(%r15)
        movq    -8(%r15), %r11
        movq    8(%r11), %rsi
        movq    %rsi, %rdi
        callq   *-16(%rbp)
        movq    %rax, %rbx
        movq    -8(%r15), %r11
        movq    16(%r11), %rsi
        movq    %rsi, %rdi
        callq   *-16(%rbp)
        movq    %rax, -16(%rbp)
        movq    free_ptr(%rip), %rsi
        movq    %rsi, %rdi
        addq    $24, %rdi
        movq    fromspace_end(%rip), %rsi
        cmpq    %rsi, %rdi
        jl block7
        movq    %r15, %rdi
        movq    $24, %rsi
        callq   collect
        jmp block6

block6:
        movq    free_ptr(%rip), %r11
        addq    $24, free_ptr(%rip)
        movq    $5, 0(%r11)
        movq    %r11, %rsi
        movq    %rsi, %r11
        movq    %rbx, 8(%r11)
        movq    $0, %rdi
        movq    %rsi, %r11
        movq    -16(%rbp), %rax
        movq    %rax, 16(%r11)
        movq    $0, %rdi
        movq    %rsi, %rax
        jmp map_vecconclusion

block7:
        movq    $0, %rsi
        jmp block6
```

Solution: The prelude should:

- Save `rbp` (1 point)
- Set `rbp` to the `rsp` (1 point)
- Save `rbx` (1 point)
- Subtract `8` from the `rsp` ($align(8 + 8) - 8 = 8$) (1 point)
- Initialize 1 slot of the rootstack and add `8` to `r15`. (2 points)
- Jump to `map_vecstart` (1 point)

The conclusion should:

- Subtract `8` from `r15` (1 point)
- Add `8` to `rsp` (1 points)
- Restore `rbx` (1 points)
- Restore `rbp` (1 points)

- Return (1 points)

```
        .align 16
map_vec:
        pushq   %rbp
        movq    %rsp, %rbp
        pushq   %rbx
        subq    $8, %rsp
        movq    $0, 0(%r15)
        addq    $8, %r15
        jmp map_vecstart

map_vecconclusion:
        subq    $8, %r15
        addq    $8, %rsp
        popq    %rbx
        popq    %rbp
        retq
```