Compilers (Python)
CSCI P423/523, Fall 2021

Midterm

Name: ____________________

This exam has 10 questions, for a total of 100 points.

1. $\boxed{\text{10 points}}$ Given the grammar below for expressions and statements, indicate which of the following programs are valid, that is, which can be parsed as a sequence of the *stmt* non-terminal.

   exp `::=` *int* `|` `input_int()` `|` `-` *exp* `|` *exp* `+` *exp* `|` *var*
   stmt `::=` `print(`*exp*`)` `|` *exp* `|` *var* `=` *exp*

   1. `-input_int() + 42`

   2. `input_int() - 42 + input_int()`

   3.
      ```
      x = input_int() * 42
      print(-x)
      ```

   4.
      ```
      y = input_int()
      x = 32 + y
      print(-x)
      ```

   5. `0`

   Solution: 2 points each

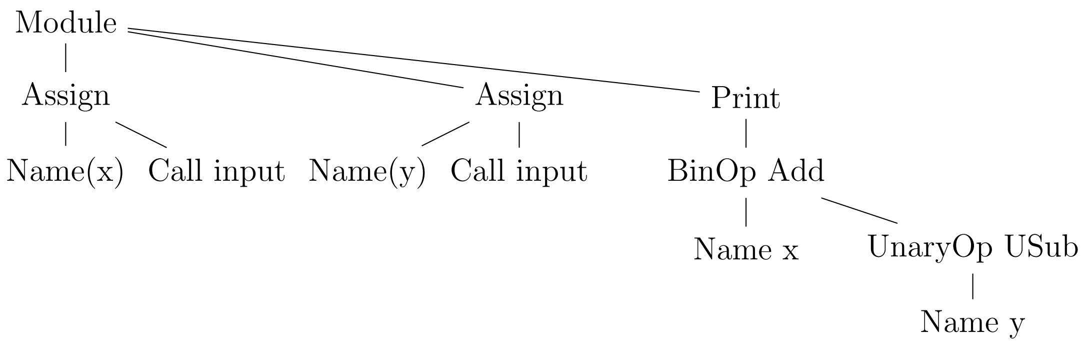
   1. Yes
   2. No, `-` takes only one argument
   3. No, `*` is not in the grammar
   4. Yes
   5. Yes

2. $\boxed{\text{12 points}}$ Convert the following program to its Abstract Syntax Tree representation and draw the tree.

```
x = input_int()
y = input_int()
print(x + -y)
```

Solution:

```
Module([Assign([Name('x')], Call(Name('input_int'), [])),
        Assign([Name('y')], Call(Name('input_int'), [])),
        Expr(Call(Name('print'), [BinOp(Name('x'), Add(),
                                        UnaryOp(USub(), Name('y')))]))])
```



3. $\boxed{\text{8 points}}$ What is the output of the following $\mathcal{L}_{\mathsf{While}}$ program? Explain the state of the x and y variable at the end of each loop iteration.

```
x = 0
y = 1
x = x + 1
while x < 4:
    y = y + x
    x = x + 1
print(y)
```

Solution: After first iteration of the loop: $x = 2, y = 2$. (2 points)

After second iteration of the loop: $x = 3, y = 4$. (2 points)

After third iteration of the loop: $x = 4, y = 7$. (2 points)

The program's output is 7. (2 points)

4. $\boxed{\text{7 points}}$ Write down the output of the Remove Complex Operands pass for the following program.

```
print(10+32 if (input_int()==0 if input_int()==1 else False)
      else 700+77)
```

Solution:

```
tmp.0 = input_int()
tmp.2 = (10+32 if ((let tmp.1 = input_int() in tmp.1==0)
                     if tmp.0==1 else False)
         else 700+77)
print(tmp.2)
```

Name: ______________________

5. $\boxed{\text{10 points}}$ Fill in the blanks to complete the following implementation of `explicate_pred` for the $\mathcal{L}_{\mathsf{If}}$ language. The `cnd` parameter is an *exp* in $\mathcal{L}_{\mathsf{If}}$; the `thn` and `els` parameters are *tail* in $\mathcal{C}_{\mathsf{If}}$. The result of `explicate_pred` must be a *tail* in $\mathcal{C}_{\mathsf{If}}$.

   You may use the other explicate functions such as `explicate_assign`. Recall that `explicate_assign` takes an *exp* in $\mathcal{L}_{\mathsf{If}}$, a variable name, and a list of statements in $\mathcal{C}_{\mathsf{If}}$, and a dictionary of basic blocks. It returns a list of statements in $\mathcal{C}_{\mathsf{If}}$.

   The auxiliary function `create_block` takes a list of statements in $\mathcal{C}_{\mathsf{If}}$ and the dictionary of basic blocks. It generates a new label, adds the label and the tail to the dictionary of basic blocks, then returns a `Goto` with the generated label.

```
def explicate_pred(cnd: expr, thn: List[stmt], els: List[stmt],
                   basic_blocks: Dict[str, List[stmt]]) -> List[stmt]:
    match cnd:
        case Name(x):
            return [If(Compare(cnd, [Eq()], [Constant(False)]),
                       [self.create_block(els, basic_blocks)],
                       [self.create_block(thn, basic_blocks)])]
        case Constant(True):
            return thn
        case Constant(False):
            return els
        case UnaryOp(Not(), operand):
            ___(a)___


        case Compare(left, [op], [right]):
            ___(b)___


        case Let(var, rhs, body):
            new_body = explicate_pred(body, thn, els, basic_blocks)
            ___(c)___


        case IfExp(test, body, orelse):
            goto_thn = create_block(thn, basic_blocks)
            goto_els = create_block(els, basic_blocks)
            new_body = explicate_pred(body, [goto_thn], [goto_els], basic_blocks)
            new_els = ___(d)___
            ___(e)___
```

Solution: 2 points each

```
(a) return explicate_pred(operand, els, thn, basic_blocks)
(b) goto_thn = create_block(thn, basic_blocks)
    goto_els = create_block(els, basic_blocks)
    return [If(cnd, [goto_thn], [goto_els])]
(c) return explicate_assign(rhs, var, new_body, basic_blocks)
(d) explicate_pred(orelse, [goto_thn], [goto_els], basic_blocks)
(e) return explicate_pred(test, new_body, new_els, basic_blocks)
```

6. $\boxed{\text{10 points}}$ Apply the Instruction Selection pass to the following $\mathcal{C}_{\mathsf{If}}$ program.

```
start:
    x4 = 1;
    tmp5 = (read);
    tmp6 = (eq? x4 tmp5);
    if (eq? tmp6 #f)
       goto block7;
    else
       goto block8;
block7:
    return 777;
block8:
    return 42;
```

Solution: 2 points each

1. Assignment of `1` to `x4`
2. Call to `(read)`
3. Assignment of `(eq?  x4 tmp5)` to `tmp6`
4. The `if` statement.
5. The two `return` statements.

```
start:
    movq $1, x4
    callq read_int
    movq %rax, tmp5
    cmpq tmp5, x4
    sete %al
    movzbq %al, tmp6
    cmpq $0, tmp6
    je block7
    jmp block8
block7:
    movq $777, %rax
    jmp conclusion
block8:
    movq $42, %rax
    jmp conclusion
```

7. 11 points Annotate each of the following instructions with the set of variables that are live immediately after the instruction. Annotate each label with the set of variables that are live before the first instruction in the label's block.

```
start:
    callq read_int
    movq %rax, x4
    callq read_int
    movq %rax, y5
    callq read_int
    movq %rax, tmp7
    cmpq $0, tmp7
    je block9
    jmp block0
block9:
    movq x4, z6
    addq $1, z6
    jmp block8

block0:
    movq y5, z6
    addq $2, z6
    jmp block8

block8:
    movq z6, %rax
    negq %rax
    jmp conclusion
```

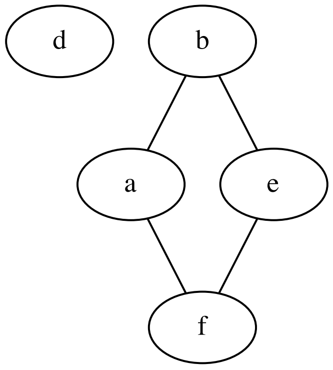
Solution: 1/2 point each

```
start:               { }
    callq read_int   { }
    movq %rax, x4    { x4 }
    callq read_int   { x4 }
    movq %rax, y5    { y5, x4 }
    callq read_int   { y5, x4 }
    movq %rax, tmp7  { y5, tmp7, x4 }
    cmpq $0, tmp7    { y5, x4 }
    je block9        { y5, x4 }
    jmp block0       { y5, x4 } or { y5 }

block9:               { x4 }
    movq x4, z6       { z6 }
    addq $1, z6       { z6 }
    jmp block8        { z6 }

block0:                   { y5 }
    movq y5, z6           { z6 }
    addq $2, z6           { z6 }
    jmp block8            { z6 }

block8:                   { z6 }
    movq z6, %rax         { }
    negq %rax             { }
    jmp conclusion        { }
```

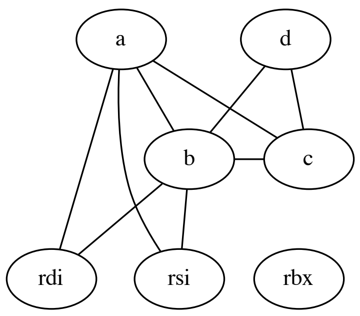
8. $\boxed{\text{8 points}}$ Consider the following results from Liveness Analysis, in which each instruction is annotated with its live-after set and each label is annotated with the live-before set of its first instruction. Draw the interference graph for this program.

```
start:              { }
    movq $1, a      { a }
    movq $42, b     { b a }
    movq b, f       { b f a }
    movq a, e       { b f e }
    addq b, e       { f }
    movq f, d       { d }
    movq d, %rax    { }
    jmp conclusion  { }
```

Solution: 2 points for each edge

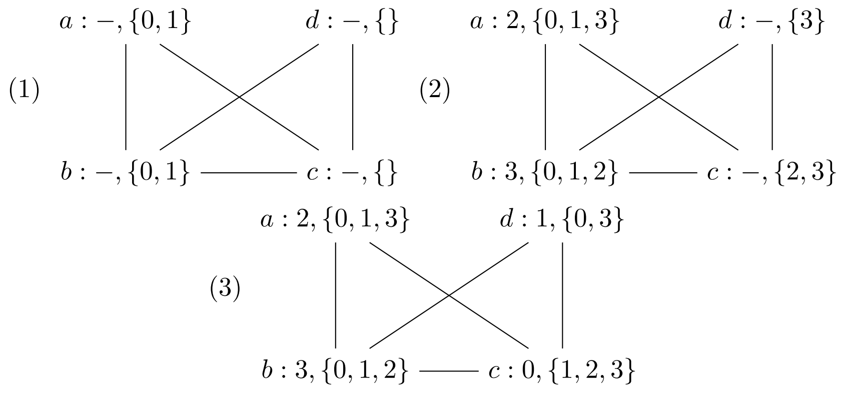


9. $\boxed{\text{14 points}}$ Given the following interference graph, use the saturation-based graph coloring algorithm to assign the variables a, b, c, and d to registers and stack locations. You may only use the registers `rdi`, `rsi`, and `rbx`. Recall that `rdi` and `rsi` are caller-saved registers and `rbx` is callee-saved. Show each step of the algorithm, include the saturation sets for each variable. To break ties regarding which variables to color first, use alphabetical order.



Solution: Here's a register to color mapping: $\{rsi : 0, rdi : 1, rbx : 2\}$.

1. We pre-color the variables with the colors of the registers that they interfere with. (2 points)
2. Both a and b have the same saturation, so we first color a to 2 (2 points) (alphabetical order) and then color b to 3 (2 points), so b is spilled to the stack at `16(%rbx)`.
3. We color c with 0 (2 points) and then d with 1 (2 points).



The assignment of variables to registers and stack locations is: (4 points)

$$\{a : \texttt{rbx}, b : -\texttt{16}(\%\texttt{rbx}), c : \texttt{rsi}, d : \texttt{rdi}\}$$

10. \boxed{10 points} Suppose that the output of the Patch Instructions pass is the following x86 assembly code. Write down the x86 assembly code for the prelude and conclusion of this program and explain your answer.

```
start:
    callq read_int
    movq %rax, %rbx
    callq read_int
    movq %rax, -16(%rbp)
    callq read_int
    movq %rax, -24(%rbp)
    callq read_int
    movq %rax, %rsi
    movq %rbx, %rdi
    addq -16(%rbp), %rdi
    movq -24(%rbp), %rbx
    addq %rsi, %rbx
    movq %rdi, %rax
    addq %rbx, %rax
    jmp conclusion
```

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
```

Solution: Note that the stack is aligned after the `pushq %rbp` instruction in the prelude. In the `start` block there are 2 spills and 1 callee-saved register is assigned to a variable, so we need $3 \times 8 = 24$ bytes of extra space on the stack, but that's not evenly divisible by 16, so we add 8 to get 32 bytes. But since we move `rsp` by 8 with the `pushq %rbx`, we only have 24 bytes left to move using the `subq`.

```
main:
    pushq %rbp
    movq %rsp, %rbp
    pushq %rbx
    subq $24, %rsp
    jmp start
conclusion:
    addq $24, %rsp
    popq %rbx
    popq %rbp
    retq
```