Compilers (Python)
CSCI P423/523, Fall 2023

Midterm

Name: ____________________

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

1. $\boxed{\text{10 points}}$ Given the following grammar, indicate which of the following programs are in the language specified by the grammar. That is, which programs can be parsed as a sequence of the *stmt* non-terminal.

```
atm  ::= int | var
exp  ::= atm | input_int() | - atm | exp if exp else exp | atm == atm | ( exp )
stmt ::= print(atm) | var = exp
```

1.
```
x = input_int()
y = 1 if x == 0 else - 2
print(y)
```

2.
```
x = 5
y = - input_int()
print(y)
```

3.
```
x = input_int()
y = 2 * x
print(y)
```

4.
```
x = input_int()
y = (2 == 3 if x == 5 else (- x if x == 0 else input_int()))
print(y)
```

5.
```
x = -5
y = 0 if -x == 5 else 1
print(y)
```

Solution: (2 points each)

1. Yes
2. No, because `input_int()` is not in *atm* but it is an argument of the negation operator.
3. No, because `*` is not in the grammar.
4. Yes
5. No, because `-x` is not in *atm* but it is an argument of the `==` operator.

2. 9 points Convert the following program to its Abstract Syntax Tree representation (see the grammar for $\mathcal{L}_{\mathsf{While}}$ in the Appendix of this exam) and draw the tree with one node per Python object.

```
sum = 0
while input_int() == 0:
  sum = (sum + 1)
print(sum)
```

Solution:

```
Module([Assign([Name('sum')], Constant(0)),
        While(Compare(Call(Name('input_int'), []), [Eq()], [Constant(0)]),
              [Assign([Name('sum')], BinOp(Name('sum'), Add(), Constant(1)))], []),
        Expr(Call(Name('print'), [Name('sum')]))])
```

3. $\boxed{\text{12 points}}$ Fill in the labeled blanks to complete the following interpreter for $\mathcal{L}_{\mathsf{Int}}$.

```
def interp_exp(e):
    match ___(a)___:
        case BinOp(left, Add(), right):
            l = interp_exp(left); r = interp_exp(right)
            return ___(b)___
        case ___(c)___:
            l = interp_exp(left); r = interp_exp(right)
            return sub64(l, r)
        case UnaryOp(USub(), v):
            return neg64(___(d)___)
        case Constant(value):
            return ___(e)___
        case Call(Name('input_int'), []):
            return input_int()

def interp_stmt(s):
    match s:
        case Expr(Call(Name('print'), [arg])):
            print(interp_exp(arg))
        case Expr(value):
            ___(f)___

def interp_Lint(p):
    match p:
        case Module(body):
            for s in body:
                interp_stmt(s)
```

Solution: (2 points each)

```
(a) e
(b) add64(l, r)                # 1 point for alternative: l + r
(c) BinOp(left, Sub(), right)
(d) interp_exp(v)
(e) value
(f) interp_exp(value)
```

4. 10 points Fill in the labeled blanks in the following implementation of the Remove Complex Operands pass.

```
def rco_exp(self, e: expr, need_atomic: bool) -> tuple[expr, Temporaries]:
    match e:
        case UnaryOp(op, operand):
            (rand, bs) = ___(a)___
            if need_atomic:
                tmp = Name(generate_name('tmp'))
                return ___(b)___, bs + [(tmp, UnaryOp(op, rand))]
            else:
                return UnaryOp(op, rand), ___(c)___
        case Name(id):
            return e, []
        ...

def rco_stmt(self, s: stmt) -> List[stmt]:
    match s:
        case Assign(targets, value):
            new_value, bs = self.rco_exp(value, ___(d)___)
            return [Assign([lhs], rhs) for (lhs, rhs) in bs] \
                 + [___(e)___]
        ...

def remove_complex_operands(self, p: Module) -> Module:
    match p:
        case Module(body):
            sss = [self.rco_stmt(s) for s in body]
            return Module(sum(sss, []))
```

Solution:

```
(a) self.rco_exp(operand, True)
(b) tmp
(c) bs
(d) False
(e) Assign(targets, new_value)
```

5. 12 points Translate the following $\mathcal{L}^{mon}_{\mathsf{While}}$ program into an equivalent $\mathcal{C}_{\mathsf{If}}$ program. The grammars for $\mathcal{L}^{mon}_{\mathsf{While}}$ and $\mathcal{C}_{\mathsf{If}}$ are in the Appendix of this exam. You may write your answer in either abstract syntax or concrete syntax.

```
Module([Assign([Name('n')], Call(Name('input_int'), [])),
        While(Begin([Assign([Name('tmp.4')], Call(Name('input_int'), []))],
                    IfExp(Compare(Name('tmp.4'), [Lt()], [Name('n')]),
                          Compare(Name('n'), [Gt()], [Constant(0)]),
                          Compare(Name('n'), [Gt()], [Constant(100)]))),
              [Expr(Constant(0))], []),
        Assign([Name('tmp.5')], Call(Name('input_int'), [])),
        Expr(Call(Name('print'), [Name('tmp.5')]))])
```

Solution:

```
start:
    n = input_int()
    goto loop.6

block.7:
    tmp.5 = input_int()
    print(tmp.5)
    return 0

block.8:
    if n > 0:
      goto loop.6
    else:
      goto block.7

block.9:
    if n > 100:
      goto loop.6
    else:
      goto block.7

loop.6:
    tmp.4 = input_int()
    if tmp.4 < n:
      goto block.8
    else:
      goto block.9
```

6. 12 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, x
  movq $1, y
  callq read_int
  movq %rax, i
  jmp loop77
loop77:
  movq i, tmp2
  cmpq $0, tmp2
  jg block79
  jmp block78

block79:
  movq y, tmp3
  movq y, tmp4
  movq tmp3, y
  addq tmp4, y
  movq i, tmp5
  movq tmp5, i
  subq $1, i
  jmp loop77
block78:
  movq y, tmp6
  movq x, %rax
  addq tmp6, %rax
  jmp conclusion
```

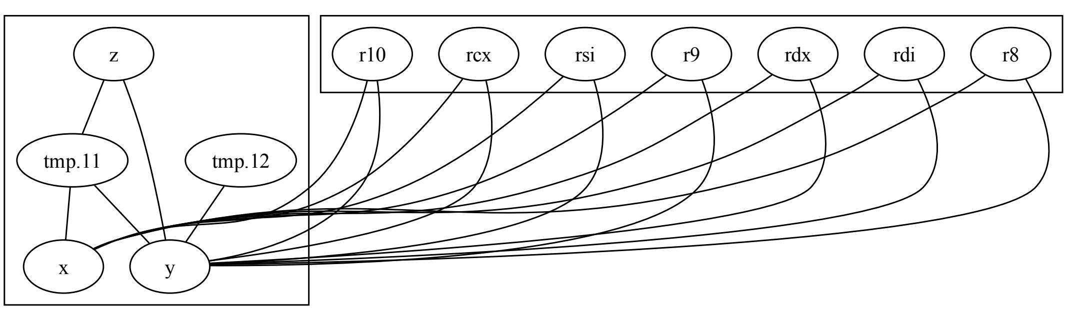
Solution: It's OK to ignore rax and rsp.

```
start:                  { rax rsp }
    callq read_int      { rax rsp }
    movq %rax, x        { rax x rsp }
    movq $1, y          { rax x y rsp }
    callq read_int      { rax x y rsp }
    movq %rax, i        { x y i rsp }
    jmp loop77          { x y i rsp }

loop77:                 { x y i rsp }
    movq i, tmp2        { x y i tmp2 rsp }
    cmpq $0, tmp2       { x y i rsp }
    jg block79          { x y i rsp }
    jmp block78         { x y i rsp }

block78:                { x y rsp }
    movq y, tmp6        { x tmp6 rsp }
    movq x, %rax        { rax tmp6 rsp }
    addq tmp6, %rax     { rax rsp }
    jmp conclusion      { rax rsp }

block79:                { x y i rsp }
    movq y, tmp3        { x y i tmp3 rsp }
    movq y, tmp4        { x i tmp3 tmp4 rsp }
    movq tmp3, y        { x y i tmp4 rsp }
    addq tmp4, y        { x y i rsp }
    movq i, tmp5        { x y tmp5 rsp }
    movq tmp5, i        { x y i rsp }
    subq $1, i          { x y i rsp }
    jmp loop77          { x y i rsp }
```

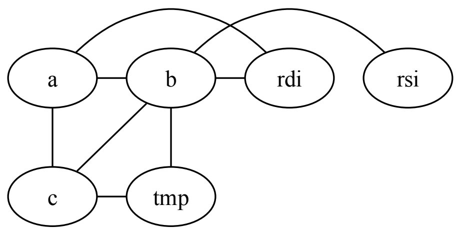
7. 12 points Given the following results from liveness analysis, draw the interference graph. (The callee and caller saved registers are listed in the Appendix of this exam. The liveness results ignore `rsp` and `rax` to simplify the graph and because they are not used in register allocation.)

```
start:
                {}
  callq read_int
                {}
  movq %rax, x
                {x}
  movq x, y
                {y, x}
  callq read_int
                {y, x}
  movq %rax, tmp.11
                {y, tmp.11, x}
  movq x, z
                {y, tmp.11, z}
  addq tmp.11, z
                {y, z}
  movq z, tmp.12
                {y, tmp.12}
  addq y, tmp.12
                {tmp.12}
  movq tmp.12, %rdi
                {%rdi}
  callq print_int
                {}
  movq $0, %rax
                {}
  jmp conclusion
                {}
```

Solution:

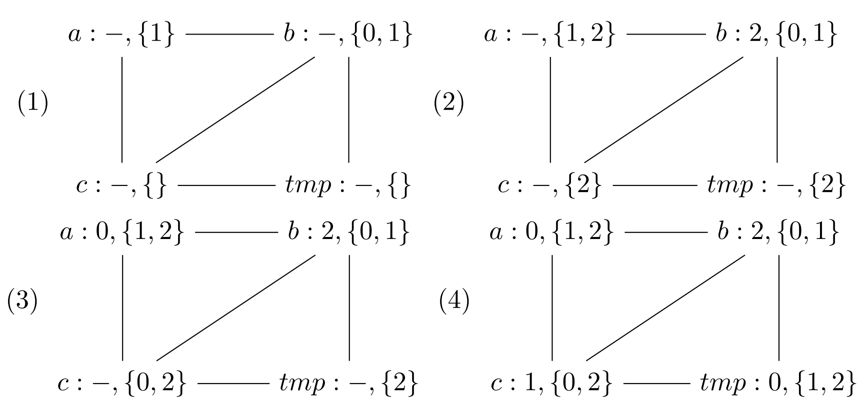


8. $\boxed{\text{14 points}}$ Given the following interference graph, use the saturation-based graph coloring algorithm to assign the variables a, b, c, and tmp to registers and stack locations. You may only use the registers `rdi` and `rsi`. Show each step of the algorithm, include the saturation sets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



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

1. We pre-color the variables with the colors of the registers that they interfere with. Variable a interferes with `rdi` (color 1) and b interferes with both `rdi` (color 1) and `rsi` (color 0).
2. b is the most saturated, so we color b to 2.
3. Now a is the most saturated, se we color a to 0.
4. Of the two remaining variables, c is the most saturated, so we color it 1.
5. Finally, we color tmp with 0.



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

$$\{a : \texttt{rsi}, b : -8(\%\texttt{rbp}), c : \texttt{rdi}, tmp : \texttt{rsi}\}$$

9. 9 points Given the following code as the result of Patch Instructions, generate a prelude and conclusion to produce a complete x86 program.

```
start:
    movq $5, -16(%rbp)
    movq $6, %rbx
    callq read_int
    movq %rax, %rdi
    movq -16(%rbp), %rcx
    addq %rbx, %rcx
    addq %rdi, %rcx
    movq %rcx, %rdi
    callq print_int
    movq $0, %rax
    jmp conclusion
```

Solution: (1 point per correct instruction)

```
        .globl main
main:
    pushq %rbp
    movq %rsp, %rbp
    pushq %rbx
    subq $8, %rsp
    jmp start

conclusion:
    addq $8, %rsp
    popq %rbx
    popq %rbp
    retq
```

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

$$
\begin{array}{rcl}
\mathit{binaryop} & ::= & \texttt{Add()} \mid \texttt{Sub()} \\
\mathit{unaryop} & ::= & \texttt{USub()} \\
\mathit{exp} & ::= & \texttt{Constant(}\mathit{int}\texttt{)} \mid \texttt{Call(Name('input\_int'),[])} \\
 & \mid & \texttt{UnaryOp(}\mathit{unaryop}\texttt{,}\mathit{exp}\texttt{)} \mid \texttt{BinOp(}\mathit{exp}\texttt{,}\mathit{binaryop}\texttt{,}\mathit{exp}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{Expr(Call(Name('print'),[}\mathit{exp}\texttt{]))} \mid \texttt{Expr(}\mathit{exp}\texttt{)} \\
\hline
\mathit{exp} & ::= & \texttt{Name(}\mathit{var}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{Assign([Name(}\mathit{var}\texttt{)], }\mathit{exp}\texttt{)} \\
\hline
\mathit{boolop} & ::= & \texttt{And()} \mid \texttt{Or()} \\
\mathit{unaryop} & ::= & \texttt{Not()} \\
\mathit{cmp} & ::= & \texttt{Eq()} \mid \texttt{NotEq()} \mid \texttt{Lt()} \mid \texttt{LtE()} \mid \texttt{Gt()} \mid \texttt{GtE()} \\
\mathit{bool} & ::= & \texttt{True} \mid \texttt{False} \\
\mathit{exp} & ::= & \texttt{Constant(}\mathit{bool}\texttt{)} \mid \texttt{BoolOp(}\mathit{boolop}\texttt{,[}\mathit{exp}\texttt{,}\mathit{exp}\texttt{])} \\
 & \mid & \texttt{Compare(}\mathit{exp}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{exp}\texttt{])} \mid \texttt{IfExp(}\mathit{exp}\texttt{,}\mathit{exp}\texttt{,}\mathit{exp}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{If(}\mathit{exp}\texttt{, }\mathit{stmt}^{+}\texttt{, }\mathit{stmt}^{+}\texttt{)} \\
\hline
\mathit{stmt} & ::= & \texttt{While(}\mathit{exp}\texttt{, }\mathit{stmt}^{+}\texttt{, [])} \\
\mathcal{L}_{\mathsf{While}} & ::= & \texttt{Module(}\mathit{stmt}^{*}\texttt{)}
\end{array}
$$

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

$$
\begin{array}{rcl}
\mathit{atm} & ::= & \texttt{Constant(}\mathit{int}\texttt{)} \mid \texttt{Name(}\mathit{var}\texttt{)} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{Call(Name('input\_int'),[])} \\
 & \mid & \texttt{UnaryOp(}\mathit{unaryop}\texttt{,}\mathit{atm}\texttt{)} \mid \texttt{BinOp(}\mathit{atm}\texttt{,}\mathit{binaryop}\texttt{,}\mathit{atm}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{Expr(Call(Name('print'),[}\mathit{atm}\texttt{]))} \mid \texttt{Expr(}\mathit{exp}\texttt{)} \\
 & \mid & \texttt{Assign([Name(}\mathit{var}\texttt{)], }\mathit{exp}\texttt{)} \\
\hline
\mathit{atm} & ::= & \texttt{Constant(}\mathit{bool}\texttt{)} \\
\mathit{exp} & ::= & \texttt{Compare(}\mathit{atm}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{atm}\texttt{])} \mid \texttt{IfExp(}\mathit{exp}\texttt{,}\mathit{exp}\texttt{,}\mathit{exp}\texttt{)} \\
 & \mid & \texttt{Begin(}\mathit{stmt}^{*}\texttt{, }\mathit{exp}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{If(}\mathit{exp}\texttt{, }\mathit{stmt}^{*}\texttt{, }\mathit{stmt}^{*}\texttt{)} \\
\hline
\mathit{stmt} & ::= & \texttt{While(}\mathit{exp}\texttt{, }\mathit{stmt}^{+}\texttt{, [])} \\
\mathcal{L}^{mon}_{\mathsf{While}} & ::= & \texttt{Module(}\mathit{stmt}^{*}\texttt{)}
\end{array}
$$

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

$$
\begin{array}{lcl}
\mathit{atm} & ::= & \texttt{Constant(}\mathit{int}\texttt{)} \mid \texttt{Name(}\mathit{var}\texttt{)} \mid \texttt{Constant(}\mathit{bool}\texttt{)} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{Call(Name('input\_int'),[])} \\
 & \mid & \texttt{BinOp(}\mathit{atm}\texttt{,}\mathit{binaryop}\texttt{,}\mathit{atm}\texttt{)} \mid \texttt{UnaryOp(}\mathit{unaryop}\texttt{,}\mathit{atm}\texttt{)} \\
 & \mid & \texttt{Compare(}\mathit{atm}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{atm}\texttt{])} \\
\mathit{stmt} & ::= & \texttt{Expr(Call(Name('print'),[}\mathit{atm}\texttt{]))} \mid \texttt{Expr(}\mathit{exp}\texttt{)} \\
 & \mid & \texttt{Assign([Name(}\mathit{var}\texttt{)], }\mathit{exp}\texttt{)} \\
\mathit{tail} & ::= & \texttt{Return(}\mathit{exp}\texttt{)} \mid \texttt{Goto(}\mathit{label}\texttt{)} \\
 & \mid & \texttt{If(Compare(}\mathit{atm}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{atm}\texttt{]), [Goto(}\mathit{label}\texttt{)], [Goto(}\mathit{label}\texttt{)])} \\
\mathcal{C}_{\mathsf{If}} & ::= & \texttt{CProgram(}\{\mathit{label}\texttt{:}\ \texttt{[}\mathit{stmt},\ldots,\mathit{tail}\texttt{]},\ldots\}\texttt{)}
\end{array}
$$