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

1. **[10 points]** Given the grammar below for expressions and statements, 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.

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

1. `print(- (10 if input_int() == 0 else 20))`

2. `x = 0
   x = -10 if input_int() == 0
   print(x + 10)`

3. `print(x = 10 if input_int() == 0 else -10)`

4. `print(---input_int() == ---10)`

5. `- input_int()`
2. [12 points] Convert the following program to its Abstract Syntax Tree representation (see the grammar for $\mathcal{L}_{\text{If}}$ in the Appendix of this exam) and draw the tree.

```python
x = 5 if input_int() == 0 else -input_int()
print(x)
```
3. [12 points] The following is a partial implementation of the type checker for expressions of the $\mathcal{L}_{\text{if}}$ language, which includes integers, Booleans, conditionals, and several primitive operations. The env parameter is a dictionary that maps every in-scope variable to a type. Fill in the blanks of this type checker.

```python
class TypeCheckLif(TypeCheckLvar):
    def type_check_exp(self, e, env):
        match e:
            case Constant(value) if isinstance(value, bool):
                return BoolType()
            case UnaryOp(Not(), v):
                t = self.type_check_exp(v, env)
                self.check_type_equal(t, BoolType())
                return ___(a)___
            case IfExp(test, body, orelse):
                test_t = self.type_check_exp(test, env)
                body_t = ___(c)___
                self.check_type_equal(body_t, ___(d)___)
                return ___(e)___
            case Begin(ss, e):
                self.type_check_stmts(ss, env)
                return ___(f)___
```

...
4. [12 points] Fill in the blanks to complete the case for `IfExp` in the following implementation of `rco_exp` (Remove Complex Operands) that translates from the $\mathcal{L}_r$ language into $\mathcal{L}_{mon}$. The grammars for these languages can be found in the Appendix of this exam.

```python
def make_begin(bs, e):
    if len(bs) > 0:
        return Begin([Assign([x], rhs) for (x, rhs) in bs], e)
    else:
        return e

class Conditionals(RegisterAllocator):
    def rco_exp(self, e: expr, need_atomic: bool) -> Tuple[expr, Temporaries]:
        match e:
            case Compare(left, [op], [right]):
                (l, bs1) = self.rco_exp(left, True)
                (r, bs2) = self.rco_exp(right, True)
                cmp_exp = Compare(l, [op], [r])
                if need_atomic:
                    tmp = Name(generate_name('tmp'))
                    return tmp, bs1 + bs2 + [(tmp, cmp_exp)]
                else:
                    return cmp_exp, bs1 + bs2
            case IfExp(test, body, orelse):
                (new_test, bs1) = ___(a)___
                (new_body, bs2) = self.rco_exp(body, False)
                (new_orelse, bs3) = self.rco_exp(orelse, False)
                new_body = ___(b)___
                new_orelse = ___(c)___
                if_exp = IfExp(___d___, new_body, new_orelse)
                if need_atomic:
                    tmp = Name(generate_name('tmp'))
                    return ___(e)___
                else:
                    return ___(f)___
                return new_body
...
5. [10 points] Translate the following $L_{\#}^{\text{mon}}$ program into $C_{\#}$. The grammar for $C_{\#}$ is in the Appendix of this exam. (The curly braces are for the concrete syntax of the $\text{Begin AST}$ node.)

```python
x = input_int()
z = ({ y = input_int()
    -y }
    if x == 0
    else input_int())
print(z)
```
6. **14 points** Given the following pseudo-x86 program, compile it to an equivalent and complete x86 program, using stack locations (not registers) for the variables. Your answer should be given in the AT&T syntax that the GNU assembler expects for .s files.

```
start:
    callq read_int
    movq %rax, x
    movq $-4, t0
    movq t0, t1
    addq x, t1
    movq t1, %rdi
    callq print_int
    movq $0, %rax
    jmp conclusion
```
7. [10 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:
    movq $0, sum
    movq $5, i
    jmp block.0

block.0:
    cmpq $0, i
    jg block.2
    jmp block.3

block.2:
    addq i, sum
    subq $1, i
    jmp block.0

block.1:
    movq sum, %rdi
    callq print_int
    movq $0, %rax
    jmp conclusion

block.3:
    jmp block.1
```
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.)

start:
  {}
callq _read_int
  {%,%rax}
movq %rax, x
  {x}
movq x, y
  {y, x}
addq $1, y
  {y, x}
movq y, z
  {y, x, z}
addq $1, z
  {y, z, x}
cmpq $0, x
  {y, x, z}
je block.1
  {y, x, z}
jmp block.2
  {y, z, x}

block.1:
  {x, z}
movq x, %rdi
  {%, %rdi, z}
callq print_int
  {z}
jmp block.0
  {z}

block.2:
  {y, z}
movq y, %rdi
  {%, %rdi, z}
callq print_int
  {z}
jmp block.0
  {z}

block.0:
  {z}
movq z, %rdi
  {%, %rdi}
callq print_int
  {}
movq $0, %rax
  {%, %rax}
jmp conclusion
  {%, %rax}
9. **10 points** Fill in the blanks to complete the following graph coloring algorithm.

```python
class PriorityQueue:
    def __init__(self, less): ...
    def push(self, key): ...
    def pop(self): ...
    def increase_key(self, key): ...
    def empty(self): ...

def color_graph(graph: UndirectedAdjList, variables: Set[location]) -> Dict[location, int]:
    unavail_colors = {}
    compare(u, v):
        return len(unavail_colors[u.key]) < len(unavail_colors[v.key])
    Q = PriorityQueue(___(a)___)
    color = {}
    for r in registers_for_alloc:
        color[Reg(r)] = register_color[r]
    for x in variables:
        adj_reg = [y for y in graph.adjacent(x) if y.id in registers]
        unavail_colors[x] = ____(b)___
    while ___(c)___:
        v = Q.pop()
        c = choose_color(v, unavail_colors)
        color[v] = c
        for u in ___(d)___:
            if u.id not in registers:
                ___(e)___
                Q.increase_key(u)
    return color
```

```
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 $\mathcal{L}_\text{if}$

| $binaryop$  | ::= $Add()$ | $Sub()$                          |
| $unaryop$   | ::= $USub()$                                    |
| $exp$       | ::= $Constant(int)$ | $Call(\text{Name}('input\_int'),[],[])$ |
|             | | $UnaryOp(unaryop, exp)$ | $BinOp(exp, binaryop, exp)$ |
| $stmt$      | ::= $Expr(Call(\text{Name}('print'),[exp]))$ | $Expr(exp)$ |

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

| $atm$      | ::= $Constant(int)$ | $Name(var)$ | $Constant(bool)$ |
| $exp$      | ::= $atm$ | $Call(\text{Name}('input\_int'),[],[])$ |
|             | | $UnaryOp(unaryop, atm)$ | $BinOp(atm, binaryop, atm)$ |
| $stmt$     | ::= $Expr(Call(\text{Name}('print'),[atm]))$ | $Expr(exp)$ |
|             | | $Assign([\text{Name}(var)], exp)$ |

Grammar for $C_\text{if}$

| $atm$      | ::= $Constant(int)$ | $Name(var)$ | $Constant(bool)$ |
| $exp$      | ::= $atm$ | $Call(\text{Name}('input\_int'),[],[])$ |
|             | | $BinOp(atm, binaryop, atm)$ | $UnaryOp(unaryop, atm)$ |
|             | | $Compare(atm, [cmp], [atm])$ | $IfExp(exp, exp, exp)$ |
|             | | $Begin(stmt^+, exp)$ |
| $stmt$     | ::= $If(exp, stmt^+, stmt^+)$ |
| $tail$     | ::= $Return(exp)$ | $Goto(\text{label})$ |
|             | | $If(Compare(atm, [cmp], [atm]), [Goto(\text{label})], [Goto(\text{label})])$ |
| $C_\text{if}$ | ::= $CProgram(\{\text{label}: [stmt,...,tail,...]\})$ |