Compilers (Python)
CSCI P423/523, Fall 2022

Midterm

Name: ____________________

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.

   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()`

   Solution: (2 points each)

   1. Yes.
   2. No, one-armed `if` is not an *exp*.
   3. No, argument of `print` must be an *exp*, not a *stmt*.
   4. Yes.
   5. No, not a *stmt*.
      ```

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

```
x = 5 if input_int() == 0 else -input_int()
print(x)
```

Solution:

```
Module([Assign([Name('x')], IfExp(Compare(Call(Name('input_int'), []), [Eq()],
                                          [Constant(0)]),
                                  Constant(5),
                                  UnaryOp(USub(), Call(Name('input_int'), [])))),
        Expr(Call(Name('print'), [Name('x')]))])
```

3. 12 points The following is a partial impelmentation of the type checker for expressions of the $\mathcal{L}_{\mathsf{if}}^{mon}$ 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.

```
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)
        self.check_type_equal(___(b)___, test_t)
        body_t = ___(c)___
        orelse_t = self.type_check_exp(orelse, env)
        self.check_type_equal(body_t, ___(d)___)
        return ___(e)___
      case Begin(ss, e):
        self.type_check_stmts(ss, env)
        return ___(f)___
      ...
```

Solution: (2 points each)

```
(a) BoolType()
(b) BoolType()
(c) self.type_check_exp(body, env)
(d) orelse_t
(e) body_t
(f) self.type_check_exp(e, env)
```

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 $\mathcal{L}_{\mathsf{If}}$ language into $\mathcal{L}_{\mathsf{if}}^{mon}$. The grammars for these languages can be found in the Appendix of this exam.

```
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)___
            ...
```

Solution: (2 points each)

```
(a) self.rco_exp(test, False)
(b) make_begin(bs2, new_body)
(c) make_begin(bs3, new_orelse)
(d) new_test
(e) (tmp, bs1 + [(tmp, if_exp)])
(f) (if_exp, bs1)
```

5. 10 points Translate the following $\mathcal{L}_{\mathsf{if}}^{mon}$ program into $\mathcal{C}_{\mathsf{If}}$. The grammar for $\mathcal{C}_{\mathsf{If}}$ is in the Appendix of this exam. (The curly braces are for the concrete syntax of the `Begin` AST node.)

```
x = input_int()
z = ({ y = input_int()
       -y }
      if x == 0
      else input_int())
print(z)
```

Solution: (Approx. 1 point per statement.)

```
start:
    x = input_int()
    if x == 0:
      goto block.4
    else:
      goto block.5

block.4:
    y = input_int()
    z = -y
    goto block.3

block.5:
    z = input_int()
    goto block.3

block.3:
    print(z)
    return 0
```

6. [14 points] Given the following psuedo-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
```

Solution:

```
        .globl main
main:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    jmp start
start:
    callq read_int
    movq %rax, -16(%rbp)
    movq $-4, -8(%rbp)
    movq -16(%rbp), %rax
    addq %rax, -8(%rbp)
    movq -8(%rbp), %rdi
    callq print_int
    movq $0, %rax
    jmp conclusion
conclusion:
    addq $32, %rsp
    popq %rbp
    retq
```

Rubric:

- prelude (3 points)
- correct use of the stack for the variables (2 points)
- call to `read_int` and move from `rax` (2 point)
- at most one memory argument per instruction (2 points)
- move to `rdi` and call to `print_int` (2 points)
- conclusion (3 points)

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.3:
    jmp block.1

block.2:
    addq i, sum

    subq $1, i

    jmp block.0


block.1:
    movq sum, %rdi

    callq print_int

    movq $0, %rax

    jmp conclusion
```

Solution:

```
block.1:
                {sum}
  movq sum, %rdi
                {%rdi}
  callq print_int
                {}
  movq $0, %rax
                {%rax}
  jmp conclusion
                {%rax}
block.2:
                {sum, i}
  addq i, sum
                {sum, i}
  subq $1, i
                {sum, i}
  jmp block.0
                {sum, i}

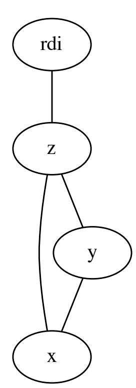
block.3:
                {sum}
  jmp block.1
                {sum}
block.0:
                {sum, i}
  cmpq $0, i
                {sum, i}
  jg block.2
                {sum, i}
  jmp block.3
                {sum, i}
start:
                {}
  movq $0, sum
                {sum}
  movq $5, i
                {sum, i}
  jmp block.0
                {sum, i}
```

8. [10 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.)

```
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}
```

Solution:



9. [10 points] Fill in the blanks to complete the following graph coloring algorithm.

```
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 = {}
    def 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] = \
            set().union([register_color[r.id] for r in adj_reg])
        ___(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
```

Solution: (2 points each)

```
(a) compare
(b) Q.push(x)
(c) not Q.empty()
(d) graph.adjacent(v)
(e) unavail_colors[u].add(c)
```

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

binaryop	::=	`Add()` | `Sub()`
unaryop	::=	`USub()`
exp	::=	`Constant(`*int*`)` | `Call(Name('input_int'),[])`
	|	`UnaryOp(`*unaryop*`,`*exp*`)` | `BinOp(`*exp*`,`*binaryop*`,`*exp*`)`
stmt	::=	`Expr(Call(Name('print'),[`*exp*`]))` | `Expr(`*exp*`)`
exp	::=	`Name(`*var*`)`
stmt	::=	`Assign([Name(`*var*`)], `*exp*`)`
boolop	::=	`And()` | `Or()`
unaryop	::=	`Not()`
cmp	::=	`Eq()` | `NotEq()` | `Lt()` | `LtE()` | `Gt()` | `GtE()`
bool	::=	`True` | `False`
exp	::=	`Constant(`*bool*`)` | `BoolOp(`*boolop*`,[`*exp*`,`*exp*`])`
	|	`Compare(`*exp*`,[`*cmp*`],[`*exp*`])` | `IfExp(`*exp*`,`*exp*`,`*exp*`)`
stmt	::=	`If(`*exp*`, `$stmt^+$`, `$stmt^+$`)`
$\mathcal{L}_{\mathsf{If}}$	::=	`Module(`$stmt^*$`)`

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

atm	::=	`Constant(`*int*`)` | `Name(`*var*`)`
exp	::=	*atm* | `Call(Name('input_int'),[])`
	|	`UnaryOp(`*unaryop*`,`*atm*`)` | `BinOp(`*atm*`,`*binaryop*`,`*atm*`)`
stmt	::=	`Expr(Call(Name('print'),[`*atm*`]))` | `Expr(`*exp*`)`
	|	`Assign([Name(`*var*`)], `*exp*`)`
atm	::=	`Constant(`*bool*`)`
exp	::=	`Compare(`*atm*`,[`*cmp*`],[`*atm*`])` | `IfExp(`*exp*`,`*exp*`,`*exp*`)`
	|	`Begin(`$stmt^*$`, `*exp*`)`
stmt	::=	`If(`*exp*`, `$stmt^*$`, `$stmt^*$`)`
$\mathcal{L}_{\mathsf{if}}^{mon}$	::=	`Module(`$stmt^*$`)`

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

atm	::=	`Constant(`*int*`)` | `Name(`*var*`)` | `Constant(`*bool*`)`
exp	::=	*atm* | `Call(Name('input_int'),[])`
	|	`BinOp(`*atm*`,`*binaryop*`,`*atm*`)` | `UnaryOp(`*unaryop*`,`*atm*`)`
	|	`Compare(`*atm*`,[`*cmp*`],[`*atm*`])`
stmt	::=	`Expr(Call(Name('print'),[`*atm*`]))` | `Expr(`*exp*`)`
	|	`Assign([Name(`*var*`)], `*exp*`)`
tail	::=	`Return(`*exp*`)` | `Goto(`*label*`)`
	|	`If(Compare(`*atm*`,[`*cmp*`],[`*atm*`]), [Goto(`*label*`)], [Goto(`*label*`)])`
$\mathcal{C}_{\mathsf{If}}$	::=	`CProgram(`$\{label\colon$ `[`$stmt, \ldots, tail$`]`$, \ldots\}$`)`