This exam has 7 questions, for a total of 100 points.

1. **10 points** Give the output of the following Python program:

   ```python
   x = [1, [2]]
y = x + x
x[0] = 3
x[1][0] = 6
print y[0] + y[1][0]
   ```

   **Solution:**
   7

2. **10 points** Give the output of the following Python program:

   ```python
   print ([0] and [1])[0] and not ([] or [1])
   ```

   **Solution:**
   False

3. **10 points** Give the output of the following Python program:

   ```python
   d = { 0: 1 }
d[d[0]] = d
i = 0
x = d
while i != 2:
x[0] = x[0] + x[0]
x = x[1]
i = i + 1
print d[0]
   ```

   **Solution:**
   4
4. **20 points** Perform heapification and closure conversion, as taught in class, on the following program to obtain an equivalent program that does not include any lambdas and in which all functions are global definitions.

```python
def c(f):
    def d(x):
        def e(y):
            return f(x,y)
        return e
    return d
print c(lambda a, b: a + b)(4)(2)
```

Write your answer using Python syntax, but using the following primitive functions:

- `create_closure` takes two arguments, a function pointer and a list of values for the free variables, and allocates a closure on the heap.
- `get_function` takes one argument, a closure, and returns the function pointer contained in the closure.
- `get_free_vars` takes one argument, a closure, and returns the list of values for the closure’s free variables.

Instead of using `let`, use a sequence of assignments.

**Solution:**

```python
def fun0(fvs, y):
    f = fvs[0]
    x = fvs[1]
    tmp = f[0]
    return get_function(tmp)(get_free_vars(tmp), x[0], y)

def fun1(fvs, x):
    f = fvs[0]
    x = [x]
    e = create_closure(fun0, [f, x])
    return e

def fun2(fvs, f):
    f = [f]
    d = create_closure(fun1, [f])
    return d

def fun3(fvs, a, b):
    return a + b

c = create_closure(fun2, [])
tmp1 = get_function(c)(get_free_vars(c), create_closure(fun3, []))
tmp2 = get_function(tmp1)(get_free_vars(tmp1), 4)
print get_function(tmp2)(get_free_vars(tmp2), 2)
```
5. **15 points** Give the LALR(1) parse table (state transition diagram) for the following grammar. For each state, list the items (grammar rules with dots) for that state and the actions (shift, goto, and reduce).

- (1) `start ::= A '.'`
- (2) `A ::= "a"`
- (3) `A ::= A "+" B`
- (4) `B ::= "b"`

**Solution:**

**state 0**

- `start ::= . A '.'`
- `A ::= . "a"`
- `A ::= . A "+" B`
  - on "a" shift to state 2
  - on A goto state 1

**state 1**

- `start ::= A . '.'`
- `A ::= A . "+" B`
  - on "+" shift to state 3
  - on "." shift to state 6

**state 2**

- `A ::= "a" .`
  - on any reduce by rule 2

**state 3**

- `A ::= A "+" . B`
- `B ::= . "b"`
  - on "b" shift to state 5
  - on B goto state 4

**state 4**

- `A ::= A "+" B .`
  - on any reduce by rule 3

**state 5**

- `B ::= "b" .`
  - on any reduce by rule 4

**state 6, accept**

- `S ::= A "." .`
6. 20 points  Translate the following AST

```
Module(Stmt([Printnl(Not(CallFunc(Name('input'), [])))]))
```

into the monomorphic intermediate representation given by the following grammar. The
only kinds of values in this subset of Python that you need to deal with are integers,
booleans (no lists or dictionaries), and pyobj (which is the tagged union of integers and
booleans). Recall that the tag for integers is 0, the tag for Booleans is 1, and the tag for
big objects is 3. You may use any of the functions from runtime.c.

```
type ::= int | bool
boolean ::= True | False
integer ::= 0 | 1 | 2 | ...
identifier = [a-zA-Z0-9_]+
value ::= boolean | integer
op ::= "==" | "!="
expr ::= Name(identifier) | Input() | Const(value)
    | Let(identifier, expr, expr) | IfExp(expr, expr, expr)
    | InjectFrom(type, expr) | ProjectTo(type, expr) | GetTag(expr)
    | Compare(expr, op, expr) | CallFunc(expr, [expr,...,expr])
stmt ::= Stmt([stmt,...,stmt]) | Printnl(expr)
module = Module(stmt)
```

Solution:

```
Module(Stmt([Printnl(InjectFrom(bool,
    Compare(Const(0), '==',
    Let(0_letify, CallFunc(Name('input_int'), [])),
    IfExp(Compare(GetTag(Name('0_letify')), '==', Const(3)),
        CallFunc(Name('is_true'), [Name('0_letify')]),
        Compare(Const(0), '!=', ProjectTo(int, Name('0_letify')))))]))
```
7. [15 points] Given the following interference graph, perform register allocation for an x86 processor, but restrict yourself to use just eax and ebx for general purpose allocation. Record your register allocation decisions in the following table. On each line, record which registers are still available for use for each variable. Also record which variable you choose to color and the chosen location.

![Interference Graph](image)

<table>
<thead>
<tr>
<th>avail. for x</th>
<th>avail. for y</th>
<th>avail. for tmp</th>
<th>variable chosen to color</th>
<th>chosen location</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Solution:** One solution is

<table>
<thead>
<tr>
<th>avail. for x</th>
<th>avail. for y</th>
<th>avail. for tmp</th>
<th>variable</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td>eax,ebx</td>
<td>eax, ebx</td>
<td>x</td>
<td>ebx</td>
</tr>
<tr>
<td>-</td>
<td>eax</td>
<td>eax</td>
<td>y</td>
<td>eax</td>
</tr>
<tr>
<td>-</td>
<td>none</td>
<td>tmp</td>
<td>-4(%ebp)</td>
<td></td>
</tr>
</tbody>
</table>

and another is

<table>
<thead>
<tr>
<th>avail. for x</th>
<th>avail. for y</th>
<th>avail. for tmp</th>
<th>variable</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td>eax,ebx</td>
<td>eax, ebx</td>
<td>x</td>
<td>ebx</td>
</tr>
<tr>
<td>-</td>
<td>eax</td>
<td>eax</td>
<td>tmp</td>
<td>eax</td>
</tr>
<tr>
<td>-</td>
<td>none</td>
<td>-</td>
<td>y</td>
<td>-4(%ebp)</td>
</tr>
</tbody>
</table>