This exam has 8 questions, for a total of 100 points.

1. **5 points** What is the output of the following Python program,

```python
x = input()
print x + input() % input(), False, True
```

given the input

-2
-1
4

**Solution:**

1 False True

2. **15 points** Draw a parse tree and an abstract syntax tree (AST) for the program in the previous question, given the grammar below and the usual precedence rules for Python. The AST should match what the Python compiler module would produce.

Grammar:

- `expression ::= identifier | integer | "True" | "False" | expression "%" expression| expression "+" expression | "input" "(" ")"
- `expression_list ::= expression | expression "," expression_list`
- `statement ::= identifier ":=" expression | "print" expression_list`
- `stmt_list ::= statement | statement stmt_list`

**Solution:**

Parse tree:
3. **5 points** Write down the definition of the `pyobj` struct used to implement the P₁ language, which contains normal integers, floats, and Booleans.

**Solution:**

```c
enum type_tag { INT, FLOAT, BOOL };  
struct pyobj_struct {  
    enum type_tag tag;  
    union {  
        int i;       /* int */  
        double f;    /* float */  
        short b;     /* bool */  
    } u;  
};  
typedef struct pyobj_struct pyobj;  
```

4. **10 points** Write down the implementation of a C function for dividing two `pyobj` for the P₁ language.

**Solution:**

```c
pyobj divide(pyobj a, pyobj b)  
{  
    if(b.type==BOOL) {  
        if(b.v.b==0) {  
            error("can’t divide by zero");  
        } else if(a.type==BOOL) {  
            return int_to_pyobj(a.v.b);  
        } else {  
            return a;  
        }  
    } else {  
```
5. 20 points Perform iterative type analysis and type specialization on the following program and write down the C code that your compiler would generate. You do not need to write down the implementations of the C structs or run-time functions that you use, but you do need to write down the function declarations (e.g., `pyobj add_pyobj(pyobj, pyobj);`).

\[
\begin{align*}
  z &= 0 \\
  y &= 1 \text{ if } \text{input()} \text{ else } 2 \\
  x &= y \text{ or } 1 \\
\text{while } z < y: \\
  &\quad z = z + x \\
  &\quad y = y - 0.5 \\
\text{print } z
\end{align*}
\]

Solution:

```c
int main() {
  pyobj y;
  pyobj x;
  pyobj z;
  z = int_to_pyobj(0);
  y = int_to_pyobj((pyobj_to_bool(input()) ? 1:2));
  x = ({ pyobj tmp0 = y; (pyobj_to_bool(tmp0) ? tmp0:int_to_pyobj(1));});
  while (pyobj_to_bool(({ pyobj tmp1 = z; less(tmp1, y)}))) {
    z = ({ pyobj left2 = z; add(left2, x);});
    y = ({ pyobj left3 = y; sub(left3, float_to_pyobj(0.5));});
  }
  print_any(z); printf("\n");
  return 0;
}
```
6. **10 points** What is the output of the following Python program?

```python
i = 0
a = {}
while i < 3:
    a[i] = lambda: i
    i = i + 1
j = 0
while j < 3:
    print a[j]()
    j = j + 1
```

**Solution:**

3
3
3

7. **20 points** Perform closure conversion on the program from the previous question, translating it to an equivalent Python program where no function has free variables.

```python
def make_closure(f, fvs):
    return [f, fvs]

def get_function(closure):
    return closure[0]

def get_fvs(closure):
    return closure[1]

def anon_0(fvs):
    i = fvs[0]
    return i[0]

i = [False]
i[0] = 0
a = {}
while i[0] < 3:
    a[i[0]] = make_closure(anon_0, [i])
    i[0] = i[0] + 1
j = 0
while j < 3:
    tmp = a[j]
    print get_function(tmp)(get_fvs(tmp))
    j = j + 1
```
8. 15 points Fill in the cases for **Name**, **PrimitiveOp**, and **Lambda** in the following function that is suppose to compute the free variables of a given AST node *n*.

```
Solution:

def free_vars(n):
    if isinstance(n, Stmt):
        return reduce(union, map(free_vars, n.nodes), set([]))
    elif isinstance(n, Discard):
        return free_vars(n.expr)
    elif isinstance(n, Const):
        return set([])
    elif isinstance(n, Name):
        return set([n.name])
    elif isinstance(n, PrimitiveOp):
        return reduce(union, map(free_vars, n.nodes), set([]))
    elif isinstance(n, Lambda):
        # the following assumes the “super lambda” approach
        # but if you didn’t use that approach, and left off the assigned_vars part, that’s ok
        return (free_vars(n.code) - set(n.argnames)) - set(assigned_vars(n.code))
    else:
        raise Exception('Error in free_vars: unrecognized AST node')
```