1. Compiler front-end (16 = 4 + 4 + 4 + 4 points)
   1a. (4 points) What is the difference between a parse tree and an abstract syntax tree (AST)?
   1b. (4 points) What kinds of errors does syntax analysis find? What kinds of errors does semantic analysis find? Give an example for each.
   1c. (4 points) What is the difference between regular expressions and context-free grammars? Which is more powerful?
   1d. (4 points) What is a “shift” in a bottom-up parser? What is a “reduce” in a bottom-up parser?

2. Scopes and symbol tables (16 = 8 + 8 points)
   Consider the following Tack program:
   
   ```tack
   f = fun () -> void {
     a = 3;
     b = 4;
     {
       a = 5;
       b := 6;
     }
     print("a == " + a + ", b == " + b + \\n\n");
   }
   
   main = fun () -> int {
     a = 1;
     f();
     print("a == " + a + \\n\n");
     -> 0;
   }
   ```

   2a. (8 points) What does the program print?
   2b. (8 points) What is the symbol table for this program?
       Your answer should show the scopes, and indicate how they are nested in each other. For each scope, list the symbols defined in that scope. You can omit the intrinsic functions, and you can omit any empty scopes.

3. Register allocation (16 points).
   Consider the following interference graph:

   Assuming that there are three registers %rax, %rdi, and %rsi available, and assuming that the spill locations on the stack are at [%rbp-8], [%rbp-16], etc., show a register assignment that minimizes spilling.
4. IR generator (18 points)
Consider the following grammar rule for a do-while loop:

\[ DW ::= do \ B \ while \ E ; \]

In this rule, \( B \) refers to a block-statement, and \( E \) refers to an expression. The do-while loop has similar semantics as in C or Java. In other words, it first executes \( B \), then evaluates \( E \). If \( E \) is true, it starts over with executing \( B \); if \( E \) is false, it goes to the next instruction after the do-while.

Write a syntax-directed definition for translating a do-while loop to IR.

\[
\begin{align*}
E.true &= \ldots \\
E.false &= \ldots \\
B.next &= \ldots \\
DW.code &= \ldots
\end{align*}
\]

5. Calling conventions (16 = 4 + 4 + 4 + 4 points)
Consider the following picture of an activation record, which illustrates the same x64/Linux calling conventions we have been using in our project:

<table>
<thead>
<tr>
<th>(Caller's AR)</th>
<th>(Callee AR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg[n-1]</td>
<td>Caller %rbp</td>
</tr>
<tr>
<td>…</td>
<td>local variables and temporaries</td>
</tr>
<tr>
<td>arg[7]</td>
<td></td>
</tr>
<tr>
<td>arg[6]</td>
<td></td>
</tr>
<tr>
<td>Return address</td>
<td></td>
</tr>
</tbody>
</table>

5a. (4 points) Which slots are written by the caller?
5b. (4 points) Which slots are written by the callee?
5c. (4 points) How does the return address get on the stack?
5d. (4 points) How does the “caller %rbp” get on the stack?

6. Data-flow analysis (18 points).
Consider the following flow graph with TACK IR:

```
ENTRY

B1: x = 1
    y = 2
    if i == 0 goto B3

B2: x = x * 3
    y = y * 3
    goto B4

B3: x = x + 4
    y = y + 4

B4: z = x + y
    return z

EXIT
```

To the right of the flow graph, there is an incomplete table with data-flow information for constant folding. Fill in the rest of the table.

```
<table>
<thead>
<tr>
<th>i</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td>in</td>
<td>NC</td>
<td>?</td>
</tr>
<tr>
<td>B2</td>
<td>in</td>
<td>out</td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td>in</td>
<td>out</td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td>in</td>
<td>out</td>
<td></td>
</tr>
</tbody>
</table>
```

Total points: 100.