1 Intermediate Representations

a. Give one reason why compilers use intermediate representations instead of translating directly from source to target language.

b. Translate each of the following Tiger programs into IR trees. Assume that all variables are temporaries unless otherwise specified.

i) \[ a[10] := b + 1 \]

ii) \[ b[2].name \], where \( b \)'s type is \{ id : int, name : string \}

iii) \[ c + 1 \], where \( c \) is the third item in the current stack frame
iv) if d < 5 then t := 3 else t := 5

v) e := "Hello world!"

vi) f := gcd(12, 16)

vii) g := myrec { field1 = 3, field2 = 5 } where myrec is defined as
type myrec = { field1 : int, field2 : int }
c. Implement the function $\text{Tr}\_\text{if}$ that translates an AST of the form $\text{if } \text{cond} \text{ then } \text{tcase} \text{ else } \text{fcase}$ into an IR tree. Use $\text{Temp\_newlabel}$ to create new labels, and $\text{T}_{\langle \text{kind} \rangle}$ to create a new node of kind $\text{kind}$.

$$\text{T\_exp } \text{Tr\_if}(\text{T\_exp } \text{cond}, \text{T\_exp } \text{tcase}, \text{T\_exp } \text{fcase})\{$$
2 Instruction Selection

Suppose we have a target language with the following tiles:

<table>
<thead>
<tr>
<th>instruction</th>
<th>tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1 = \text{add}(r_2, c)$</td>
<td>Plus Const, Plus Const</td>
</tr>
<tr>
<td>$r_1 = r_2$</td>
<td>Move Temp</td>
</tr>
<tr>
<td>$r_1 = \text{load}(r_2 + c)$</td>
<td>Mem Const, Mem Const, Mem Const, Mem Const</td>
</tr>
<tr>
<td>store($r_1 + c$, $r_2$)</td>
<td>Move Move Move Move</td>
</tr>
</tbody>
</table>

a. Tile each of the following trees using Maximal Munch (circling the tiles), and write the corresponding assembly program.

i) Move
   
   ```plaintext
   Mem 5
   \|___
   Plus
   \|___
   a 1
   ```

ii) Move
    
    ```plaintext
    Temp Plus
    \|___
    b 1 Plus
    \|___
    2 Plus
    \|___
    3 Mem
    \|___
    Plus
    \|___
    a 1
    ```
b. Suppose you were implementing the Maximal Munch algorithm for this target language. Fill in the Mem case of `munchExp`, in the style of the provided code for the Const case. You may assume that the tree IR contains a node type `T.PLUS` for addition.

```c
Temp_temp munchExp(T_exp e){
    char buf[32];
    switch(e->kind){
        case T_CONST:
            Temp_temp r = Temp_newtemp();
            emit("d0 = add(zero, %d)", r, e->u.CONST);
            return r;
        case T_MEM:
            // Mem case implementation
    }
}
```
3 Dataflow Analysis

a. Write or draw the live range for each variable in the following program.

```
1: a = fp + 1
2: b = [a]
3: c = b + 1
4: d = [a] + 3
5: e = c + d
```

b. Draw the control flow graph for the following program:

```
test:
cjmp (x < 0) done
x = x - y
y = y + 1
jump test
done:
z = y
```

c. The rules for liveness analysis are as follows:

\[
in[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \\
\text{out}[n] = \bigcup_{n' \in \text{succ}(n)} \text{in}[n']
\]

Perform liveness analysis on the CFG above, computing the in and out sets for each node until you reach a fixed point. You only need to show your final results, but if you show each iteration you have a higher chance of receiving partial credit.
4 Register Allocation

a. Why do we need to do register allocation on the output of instruction selection?

b. Consider the following program, annotated with live variable information:

```plaintext
// live: {b, c, e}
\texttt{a = b + c}
// live: {a, c, e}
\texttt{d = a - e}
// live: {a, c, d, e}
\texttt{f = d + c}
// live: {a, e, f}
\texttt{g = fun(f, a, e)}
// live: {g}
```

i) Draw the interference graph for the program.

ii) Can the graph be 3-colored without spilling? If not, why not?

iii) 3-color the graph using Kempe’s algorithm. Indicate the order in which you removed nodes from the graph, and the color assigned to each node (you can use the numbers 1, 2, and 3 for the colors). Mark any spilled nodes with an X.
c. Consider the following program, annotated with live variable information:

```
// live: {b, c}
a = b + c
// live: {a, c}
d = c
// live: {a, d}
e = [d] + a
// live: {e}
```

i) Draw the interference graph for the program, connecting any move-related nodes with a dotted edge.

ii) Pick a pair of move-related nodes from your graph. If they were coalesced into a single node, what would be the degree of the resulting node?