### 1. Intermediate Representations

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

There are a lot! Smaller steps are easier to get right; IRs can be designed to make certain aspects of the program easier to analyze and optimize; the translation from source to IR can be reused for multiple target languages.

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

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

```
Move
  Mem
  Plus
    Temp
    10 * W
    1

Mem
  Plus
    b
    Const

Mem
  Plus
    Temp
    10 * W
    a
```

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

```
Mem
  Plus
    Temp
    Const
    1 * W

Mem
  Plus
    b
    Const
    2 * W
```

iii) $c + 1$, where $c$ is the third item in the current stack frame

```
Plus
  Mem
  Const
  1

Mem
  Plus
    Temp
    Const
    fp
    2 * W
```
iv) if \( d < 5 \) then \( t := 3 \) else \( t := 5 \)

v) \( e := "Hello world!" \)

vi) \( f := \text{gcd}(12, 16) \)

vii) \( g := \text{myrec} \{ \text{field1} = 3, \text{field2} = 5 \} \) where \text{myrec} is defined as

\[
\text{type myrec} = \{ \text{field1} : \text{int}, \text{field2} : \text{int} \}
\]
c. Implement the function `Tr_if` that translates an AST of the form `if cond then tcase else fcase` into an IR tree. Use `TempNewLabel` to create new labels, and `T_{kind}` to create a new node of kind `kind`.

```c
T_exp Tr_if(T_exp cond, T_exp tcase, T_exp fcase){
    Temp_label t = TempNewLabel();
    Temp_label f = TempNewLabel();
    Temp_label join = TempNewLabel();

    return T_Seq(T_Cjmp(T_ne, cond, T_Const(0), t, f), T_Seq(
        T_Seq(T_Label(t), T_Seq(tcase, T_Jump(join))), T_Seq(
            T_Seq(T_Label(f), T_Seq(fcase, T_Jump(join))), T_Label(join))));
}
```
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, Plus, Const, 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, Mem, Mem, Mem, Plus, Plus, Const, Const</td>
</tr>
<tr>
<td>$\text{store}(r_1 + c, r_2)$</td>
<td>Move, Move, Move, Move, Mem, Mem, Mem, Mem, Plus, Plus, Const, Const</td>
</tr>
</tbody>
</table>

a. Tile each of the following trees using Maximal Munch (circling the tiles), and write the corresponding assembly program. Assume that there exists a register called “zero” that always holds the value 0.

i)  

```
Move
  Mem
    Plus
      a
```

$t = \text{add}(\text{zero}, 5)$

```
store(a + 1, t)
```

ii)  

```
Move
  Temp
    b
      1
        Plus
          2
            Plus
              3
                Mem
                  Plus
                    a
```

$t_1 = \text{load}(a + 1)$

```
t_2 = 3 + t_1
t_3 = 2 + t_2
t_4 = 1 + t_3
b = t_4
```
\[ t_1 = \text{add}(\text{zero}, 4) \]
\[ t_2 = \text{load}(t_1 + 2) \]
\[ t_3 = \text{add}(\text{zero}, 5) \]
\[ \text{store}(t_2 + 3, t_3) \]

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 = \text{add}(\text{zero}, %d)\), r, e->u.CONST);
            return r;
        case T_MEM:
            Temp_temp r = Temp_newtemp();
            switch(e->u.MEM->kind){
                case T_CONST:
                    emit("d0 = \text{load}(\text{zero} + %d)\), r, e->u.MEM->u.CONST);
                    return r;
                case T_PLUS:
                    T_exp plus = e->u.MEM->u.PLUS;
                    if(plus->left->kind == T_CONST){
                        emit("d0 = \text{load}(s0 + %d)\), r, munchExp(plus->right), plus->left->u.CONST);
                        return r;
                    } else if(plus->right->kind == T_CONST){
                        emit("d0 = \text{load}(s0 + %d)\), r, munchExp(plus->left), plus->right->u.CONST);
                        return r;
                    }
                default:
                    emit("d0 = \text{load}(s0 + 0)\), r, munchExp(e->u.MEM));
                    return r;
            }
        default:
            emit("d0 = \text{load}(s0 + 0)\), r, munchExp(e->u.MEM));
            return r;
    }
}
```
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

a: 1–4
b: 2–3
c: 3–5
d: 4–5
e: 5–

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

1: cjmp x < 0
4: z = y
2: x = x - y
3: y = y + 1

1. cjmp x < 0

2. x = x - y

3. y = y + 1

4. z = y

5. done:

c. The rules for liveness analysis are as follows:

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

\[ \text{out}[n] = \bigcup_{n' \in \text{succ}(n)} \text{in}[n'] \]

Perform liveness analysis on the CFG above, computing the \text{in} and \text{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.

<table>
<thead>
<tr>
<th>Iteration 1:</th>
<th>Iteration 2:</th>
<th>Iteration 3:</th>
<th>Result:</th>
</tr>
</thead>
<tbody>
<tr>
<td>in[3] = {x, y}</td>
<td></td>
<td></td>
<td>in[3] = {x, y}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>in[4] = {y}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>out[4] = {}</td>
</tr>
</tbody>
</table>
4 Register Allocation

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

Instruction selection generates a new temporary register for each subexpression, which will almost certainly result in more temporary registers than there are machine registers.

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

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

i) Draw the interference graph for the program.

![Interference graph for the program](image)

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

No: there are more than 3 variables live at once. (In the graph, this corresponds to a subgraph of 4 nodes that all have degree at least 3.)

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.

There are many possible orders and colorings. One is:

Nodes removed: b, f, g, a (marked for spilling), c, d, e

Resulting graph:

![Resulting graph with colors](image)
c. Consider the following program, annotated with live variable information:

```plaintext
// 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.

```
    a   e
   / \  /
 /   /   \\
 b  c    d
```

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?

If c and d were coalesced, the degree of the resulting node would be 2. The graph would be:

```
    a   e
   /   /
 /     \\
 b  c,d
```