1 Helpful conventions

You may find the following conventions and starting points helpful:

1. All variables are stored on the stack at the beginning of the program.
2. $29$ should point to just below the first variable on the stack, so that the first variables offset from $29$ is $4$.
3. If labels are called elseifi and endifi where $i \ 0, 1, \ldots$ is different for every if.

2 Extending WLP4: Pre- and Post-Increment

In the previous tutorial that we added pre- and post-increment operators for WLP4’s ints. These operators used the grammar rules below. We will extend these operators to work for pointers as well.

\[
\text{factor} \rightarrow \text{PLUS PLUS} \ \text{lvalue} \\
\text{factor} \rightarrow \text{lvalue} \ \text{PLUS PLUS}
\]

1. Give the type rules for the above grammar rules.
2. Show how to modify the code generation for these rules in the last tutorial so that the operators can either increment int or int* variables.

3 Extending WLP4: Pointers as Conditions

Recall that in C, pointers can be used as conditions, such as in the following example:

```c
int *c = NULL;
if (c) {
    //c is not null
} else {
    //c is null
}
```

This is most commonly used to check whether pointers are null.

1. Modify the WLP4 grammar so that pointers can be used in if and while tests.
2. Describe how to modify the WLP4 type checker to account for these new rules.
3. Write pseudocode to generate code for these new rules.
4 Procedures in WLP4

4.1 Helpful conventions

You are given the following WLP4 program that runs on a MIPS machine:

```c
int f( int *a, int b, int *c, int d){
    int e = 20;
    int *f = NULL;
    return 9;
}

int wain( int a, int b){
    a = f(&a, 1, &b, 2);
    return 0;
}
```

Before the program start executing line 8 of the program, the machine has the following state of registers and memory:

$n01 = 0x00000020 \quad n02 = 0x00000002 \quad n03 = 0x00000000 \quad n04 = 0x00000004$
$n05 = 0x00000088 \quad n06 = 0x00000000 \quad n07 = 0x00000000 \quad n08 = 0x00000000$
$n09 = 0x00000000 \quad n10 = 0x00000000 \quad n11 = 0x0000000B \quad n12 = 0x00000000$
$n13 = 0x00000000 \quad n14 = 0x00000000 \quad n15 = 0x00000000 \quad n16 = 0x00000000$
$n17 = 0x00000000 \quad n18 = 0x00000000 \quad n19 = 0x00000000 \quad n20 = 0x00000000$
$n21 = 0x00000000 \quad n22 = 0x00000000 \quad n23 = 0x00000000 \quad n24 = 0x00000000$
$n25 = 0x00000000 \quad n26 = 0x00000000 \quad n27 = 0x00000000 \quad n28 = 0x00000000$
$n29 = 0x00fffffc \quad n30 = 0x00fffff8 \quad n31 = 0x8123456c$

<table>
<thead>
<tr>
<th>address</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td></td>
</tr>
<tr>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x00000004</td>
<td></td>
</tr>
<tr>
<td>0x00000008</td>
<td></td>
</tr>
<tr>
<td>0x0000000c</td>
<td></td>
</tr>
<tr>
<td>0x00000010</td>
<td></td>
</tr>
<tr>
<td>0x00000014</td>
<td></td>
</tr>
<tr>
<td>0x00000018</td>
<td></td>
</tr>
<tr>
<td>0x0000001c</td>
<td></td>
</tr>
<tr>
<td>0x00000020</td>
<td></td>
</tr>
<tr>
<td>0x00000024</td>
<td></td>
</tr>
<tr>
<td>0x00000028</td>
<td></td>
</tr>
<tr>
<td>0x0000002c</td>
<td></td>
</tr>
<tr>
<td>0x00000030</td>
<td></td>
</tr>
<tr>
<td>0x00000034</td>
<td></td>
</tr>
<tr>
<td>0x00000038</td>
<td></td>
</tr>
<tr>
<td>0x0000003c</td>
<td></td>
</tr>
</tbody>
</table>

Now, assume the machine has finished executing the procedure f and is ready to return (has not yet restored the registers and stack). Give the memory blocks diagram for the stack memory and the values stored in register $n3, n29, and n30. You can assume that the compiler uses the callee-saves approach when saving/restoring registers. Also note that the only register f modifies is $5.$