1 Pre- and Post-Increment Code Generation

1.1 Pre-Increment Code Generation

First recall what these two operators do. \texttt{++i} increases the value of \texttt{i} by 1 and then returns the new value, while post-increment increases the value of \texttt{i} by 1 but returns the old value.

Pre-increment is a bit more straightforward:

```
void genCode(tree t)
    ...
    if(t.rule is "factor -> PLUS PLUS lvalue"){
        // generate code to put the lvalue in $3
        genCode(t.children[2])
        // $3 contains an lvalue, which means we need to fetch the actual value
        // from memory
        // We're making the assumption that the full address and not just an offset
        // Otherwise we need to do something like: add/sub $3, $3, $29
        lw $5, 0($3)
        // Add one to the value
        add $5, $5, $11
        // Save the new value
        sw $5, 0($3)
        // Return new value
        add $3, $5, $0
    }
    }
```

Note that this is a factor and not an lvalue, so we don’t need to return a memory address. This also means we can’t nest increments (for example, \texttt{++++i}).
1.2 Post-Increment Code Generation

Post-increment requires that we remember the old value:

```c
void genCode(tree t)
...
if(t.rule is "factor -> lvalue PLUS PLUS"){
    // generate code to put the lvalue in $3
    // You'll do this in your assignment
    genCode(t.children[0])
    // $3 contains an lvalue, which means we need to fetch the actual value
    // from memory
    // We're making the assumption that the full address and not just an offset
    // Otherwise we need to do something like: add/sub $3, $3, $29
    lw $5, 0($3)
    // Copy the old value
    add $6, $5, $0
    // Add one to the value
    add $5, $5, $11
    // Save the new value
    sw $5, 0($3)
    // Return old value
    add $3, $6, $0
}
```

As you can see, post-increment is not much more expensive than pre-increment for integers. It might be considerably more expensive for more complex data structures, though.
2 Switch Statement Code Generation

The biggest challenge here is remembering which label the case statements need to jump to when they finish execution. We do this by augmenting the tree with a parentLabelId field on each node.

```c
void genCode(tree t)
...
if(t.rule is "statement -> SWITCH LPAREN expr RPAREN LBRACE cases default RBRACE"){
    x = genLabelID()
    // Evaluate the expr and push it onto the stack
    genCode(t.children[2])
    push($3) // put expr onto to compare with each case
    // Pass the label ID we generated to the children
    c.children[5].parentLabelId = x
    // Generate all the case statements
    genCode(t.children[5])
    // Generate code for the default case
    genCode(t.children[6])
    endSwitch + x:
}
if(t.rule is "cases -> cases case"){
    // Generate the code for the case statements
    // Pass on the parentLabelId
    t.children[0].parentLabelId = t.parentLabelId
    genCode(t.children[0])
    t.children[1].parentLabelId = t.parentLabelId
    genCode(t.children[1])
}
if(t.rule is "cases -> "){
}
}
if(t.rule is "case -> CASE LPAREN expr RPAREN LBRACE statements RBRACE"){
    x = genLabelID()
    // Evaluate the expr
    genCode(t.children[2])
    // Pop the switch statement expression from the stack
    pop($5)
    // Compare the case statement and switch statement expressions
    bne $3, $5, endLabel + x
    genCode(t.children[5])
    lis $12
    .word endSwitch + t.parentLabelId
    jr $12
    endLabel + x:
    push($5)
}
if(t.rule is "default -> DEFAULT LBRACE statements RBRACE") {
    pop($5) // throw away expr, as it isn't needed anymore
    genCode(t.children[2])
}
}
```

As an exercise, reduce the number of pushes and pops required to generate a switch statement. You will still want to push the expression’s result onto the stack at the beginning of the switch (why?).