Lab 5: Recursions

Submission timestamps will be checked and enforced strictly by the CourseWeb; **late submissions will not be accepted**. Check the due date of this lab on the CourseWeb. Remember that, per the course syllabus, if you are not marked by your recitation instructor as having attended a recitation, your score will be cut in half.

In this lab, you are going to write recursive functions (procedures) and make sure that they follow all calling conventions.

Recursive Functions

Simply put, a recursive function is a function that call itself. Consider a Java method shown below:

```
public int aRecFunct(....)
{
    :
    x = aRecFunct(....);
    :
    return ...
}
```

Since the number of times that a recursive function will call itself cannot be determined before the program is assembled, there should be only one copy of the assembly code of a recursive function but keep jumping back (`jal`) over and over. Let's look at a classic problem, factorial. In mathematic, $n!$ (n factorial) is defined as:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

where $0! = 1$ and for simplicity $n!$ where $n < 0$ is undefined. For example, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$. If you are asked to write a program to calculate the factorial of a number, most likely you will use some kind of loop statement (`for`, `while`, or `do`).

Let's look at the definition of $n!$ above closely. There is an equal symbol where the left side of the equal symbol is our problem ($n!$) and on the right side is our solution ($n \times (n-1) \times \cdots \times 1$). Note that this solution does not contain any of our original problem namely factorial. So, according to the solution above, it is not suitable for recursion.

Now, consider $(n-1) \times (n-2) \times \cdots \times 1$, it is actually $(n-1)!$. So, we can rewrite the above definition of factorial as

$$n! = n \times (n-1)!$$

Note that the right side of the equal symbol (solution) contains our original problem (factorial). In other words, the above definition says that to find out what is $n!$, you must first calculate the value of $(n-1)!$ and multiplied by n. So, if you write a function that calculates the value of a factorial based on the above definition, your factorial function must call itself as shown below:

```
public int factorial(int n)
{
    if(n == 0)
        return 1;
```

```
    else
        return n * factorial(n - 1);
}
```

It is important that a recursive function must have one or more base cases which do not make any recursive call. In the code above, the base case is when $n == 0$. The function simply returns 1 without making any recursive calls. A recursive function should have one or more recursive cases which call itself (recursive call). However, every time a recursive call is made, it should bring the function closer to the base case. Otherwise, your function may keep calling itself until you run out of memory.

Now, let's think about the `_factorial` function in MIPS assembly instead of Java. First, the main program will call the `_factorial` function. So, the `_factorial` function is a **callee**. However, the `_factorial` function will call itself (recursive call). Therefore, the `_factorial` function is also a **caller**. Do you remember the rule of sharing registers?

1. Caller should not expect that values stored in `$t0` to `$t9` will be maintained across a function call
2. Callee must maintain values stored in `$s0` to `$s7`.

As usual, `jal` must be used to call a function, `jr $ra` must be used to return to caller, arguments should be in `$a0` to `$a3`, and return values should be in `$v0` or `$v1`. The `_factorial` function needs only one argument and returns a value. So, a version of MIPS assembly of the `_factorial` function could be:

```
_factorial:
        addi  $sp, $sp, -8              # Allocate activation frame
        sw    $s0, 4($sp)               # Backup $s0
        sw    $ra, 0($sp)               # Backup $ra
        add   $s0, $zero, $a0           # $s0 is n
        beq   $s0, $zero, returnOne     # Check whether n == 0
        addi  $a0, $s0, -1              # $a0 = n - 1
        jal   _factorial                # Calculate (n - 1)!
        multu $s0, $v0                  # Calculate n * (n - 1)!
        mflo  $v0                       # Set return value to n * (n - 1)!
        j     return
returnOne:
        addi  $v0, $zero, 1             # Set return value to 1
return:
        lw    $s0, 4($sp)               # Restore $s0
        lw    $ra, 0($sp)               # Restore $ra
        addi  $sp, $sp, 8               # Deallocate activation frame
        jr    $ra                       # Go back to caller
```

To calculate the value of a factorial, simply set `$a0` to n and call the `_factorial` function, and the factorial value will be in `$v0`.

```
        addi  $a0, $zero, 5             # What is 5!
        jal   _factorial
        add   $s0, $zero, $v0           # $s0 = 5!
```

What to do?

For this lab, you have to write three recursive functions as follows:

1. The `_sum` function where $sum(n) = n + (n-1) + \cdots + 1$. Note that from the definition of the $sum()$ function, we can think of it as $sum(n) = n + sum(n-1)$ where $sum(0) = 0$ and for simplicity, $sum(n)$ where n is a negative integer is undefined. This function should take exactly one argument (n) in `$a0` and return the value of $sum(n)$ in `$v0`.

2. The `_pow` function where $pow(x, y) = x^y$. Similarly to the $sum()$ function, the definition of the $pow()$ function can be defined recursively as $pow(x, y) = x \times pow(x, y-1)$ where $pow(x, 0) = 1$ and for simplicity $x \geq 0$ and $y \geq 0$. This function should take two arguments x in `$a0` and y in `$a1` and return the value of $pow(x, y)$ in `$v0`. For this function, you are allowed to use the multiplication instruction `multu`.

3. The `_fibonacci` function (F) where $F(0) = 0$, $F(1) = 1$, and $F(n) = F(n-1) + F(n-2)$ where $F(n)$ where $n < 0$ is undefined. For example,

$$\begin{aligned} F(4) &= F(3) + F(2) \\ &= (F(2) + F(1)) + (F(1) + F(0)) \\ &= ((F(1) + F(0)) + F(1)) + (F(1) + F(0)) \\ &= ((1 + 0) + 1) + (1 + 0) \\ &= 3 \end{aligned}$$

This function should take exactly one argument (n) in `$a0` and return the value of $F(n)$ in `$v0`.

The starter code (`lab05.asm`) is given on the CourseWeb. This starter code asks a users to enter a series of numbers and show values of summation, power, and Fibonacci as shown below:

```
Summation: sum(n)
Please enter an integer (greater than or equal to 0): 5
sum(5) is 15.
Power: pow(x,y)
Please enter an integer for x (greater than or equal to 0): 2
Please enter an integer for y (greater than or euual to 0): 4
pow(2,4) is 16.
Fibonacci: F(n)
Please enter an integer (greater than or equal to 0): 5
F(5) is 5.
```

Note that you will find all functions that you must implement in the starter code right after the main program. For simplicity, we will not test your program with negative values.

Submission

Submit your `lab05.asm` file via CourseWeb before the due date stated on the CourseWeb.