Compiling Functions

Consider the following procedure and main program:

```
main() {

 int x=3;
 x = leaf_example(10,2,3,x);

}

int leaf_example (inf g, int h , int i , int j)
{
  int f;
  f = (g+h) - (i+j);
  return f;
}
```


How does it get compiled?

Assume that `x` is stored in `$s0`

```
main:
 addi $s0,$zero,3
 addi $a0,$zero,10
 addi $a1,$zero,2
 addi $a2,$zero,3
 add  $a3,$zero,$s0
 jal leaf_example
return_address:
 addi $s0,$zero,$v0
```

This puts the parameters in the registers `$a0`,`$a1`,`$a2`,`$a3` and calls the procedure leaf_example `jal` puts the reuturn address in register `$ra` and jumps to `leaf_example`.


```
addi $sp,$sp,-4  #make room on the stack for one item.
sw  $s0,0($sp) #store register $s0 on the top of the stack.
add $t0,$a0,$a1
add $t1,$a2,$a3
sub $s0,$t0,$t1
add $v0,$s0,$zero #move $s0 to the return register $v0
lw $s0,0($sp)
addi $sp,$sp,4
jr $ra
```


How are stacks implemented on a MIPS processor?

Unless you're writing an operating system you can assume that the register $sp points to the top of the stack.

Thus to read the value on the top of the stack you use :

```
lw $s0,0($sp)
```

To store another element on the stack first need to make room:

```
addi $sp,$sp,-4
```

and then store the element on the stack:

```
sw $s0,0($sp)
```

If you want to remove the top element from the stack you just change the stack pointer:

```
addi $sp,$sp,4
```

Reading and removing an element from the top of a stack is called *poping* an element.

Adding an element to the top of the stack is called *pushing* an element to the top of the stack.

You must not interfere with values on the stack which are not yours. (Why?)

Recursive functions

```
int fact (int n)
 {
   if (n < 1)
     return (1);
   else
     return (n*fact(n-1));
 }
```


The parameter `n` corresponds to the argument register `$a0`.

```
fact:
 addi $sp,$sp,-8 # adjust the stack 2 items.
 sw  $ra,4($sp) #save the return address
 sw  $a0, 0($sp) #save the argument n

 slti $t0,$zero,1 #test if n < 1
 beq  $t0,$zero,L1 #if n>=1 goto L1
 addi $v0,$zero,1 #return 1
 lw  $ra,4($sp)
 lw  $a0,0($sp)
 addi $sp,$sp,8 # pop 2 items off the stack
 jr $ra
```



```
L1:
 addi $a0,$a0,-1 # if n>=1 argument gets (n-1)
 jal fact #call the fact with (n-1)
```

Now $v0 contains the the result of `fact(n-1)`

```
  lw $a0,0($sp) #restore argument n
  lw $ra,4($sp) #restore the return address
  addi $sp,$sp,8 #pop two items.
  mult $v0,$a0,$v0 # return n*fact (n-1)
  jal $ra
```

```
 sum:
   addi $sp,$sp,-4 #Make room for one element on the stack.
   sw $ra, 0($sp) #Store the return address on the stack.
   beq $a0,$zero,acc_zero #If n=0 goto acc_zero
   add $a1,$a1,$a0 # $a1 = acc + n
   addi $a0,$a0,-1  # $a0 = $a0 -1
   jal sum # call sum again. ($v0 contains the return value)
   lw $ra,0($sp) #restore $r0 from the stack.
   addi $sp,$sp,4 # reset the stack pointer.
   jal $ra #return.
acc_zero:
   addi $v0,$zero,$a1
   lw $ra,0($sp)
   add $sp,$sp,4
   jr $ra
```


Revision instructions you should know about: `addi`,`add`,`sub`, `lw`, `sw` , `beq` , `bne` , `slt` , `j` , `jr`, `jal` , `mult`, `lui` , `ori` .

Each instruction is very simple and not very hard to learn.

There are some subtleties with arithmetic concerning negative numbers, overflow and underflow. These we will look at soon.

Another example:

```
int sum(int n , int acc) {
 if (n>0)
  return sum (n-1,acc + n);
 else
 return acc;
}
```


How can we make this code more efficient?

```
 sum:
  beq $a0,$zero,sum_exit
  addi $a1,$a1,$a0
  addi $a4,$a0,-1
  j sum
sum_exit:
  add $v0,$a1,$zero
  jr $ra
```

We have got rid of the stack in this example. We have achieved this by observing that the only thing that was pushed on the stack was the address `sum`.

Such an optimisation is called elimination of tail recursion.

Now implement the following more efficient version:

```
int fib_iter (int a, int b , int count) {
 if (count == 0)
   return b;
 else
  return fib_iter(a+b, a, count -1);
```


Exercise implement the following function:

```
int fib(int n){
  if (n==0)
   return 0;
  else if (n==1)
   return 1;
  else
   return fib(n-1) + fib(n-2);
}
```