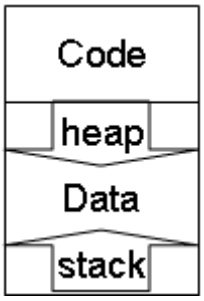
Lab: Memory and Subroutines

Background

Before you begin watch the video on implementing recursive factorial on Schoology. We will divide the MIPS data portion of memory into two parts: the heap and the stack. We'll store any data structures (strings, arrays, objects, etc) on the heap. But we will be primarily concerned with the stack, which we'll use to handle calls to subroutines.



By convention, the register `$sp` will always contain the address of the top element on the stack. We'll therefore represent *push* and *pop* with the following code segments.

```
subu $sp, $sp, 4
sw reg, ($sp)
```

push

```
lw reg, ($sp)
addu $sp, $sp, 4
```

pop

By convention, whenever we call a subroutine, we'll place its arguments in registers `$a0`-`$a3` (and any additional ones on the stack), and the subroutine will leave its return value in register `$v0`. We will use the `jal` instruction to jump to a subroutine. This instruction stores the return address in register `$ra`. We will therefore need to save whatever value was in `$ra` to the stack before calling a subroutine. At the end of a subroutine, we'll jump back to the address in this register. Here is a simple call to subroutine `square`, and the corresponding definition of `square`.

```
subu $sp, $sp, 4
sw $ra, ($sp)
jal square
lw $ra, ($sp)
addu $sp, $sp, 4
```

call to subroutine `square`

```
square:
  mult $a0, $a0
  mflo $v0
  jr $ra
```

definition of subroutine `square`

In general, here are the steps that the caller (code calling the subroutine) and callee (the subroutine) must follow.

Caller: before calling the subroutine

- Save to the stack any register values that will be needed later (including `$ra`).
- Place argument values in `$a0`-`$a3` (and additional ones on stack).
- Use `jal` to call subroutine.

Callee: end of subroutine

- Place return value in `$v0`.
- Use `jr $ra` to return from subroutine.

Caller: after return from subroutine

- Restore any saved register values from stack (including `$ra`) in reverse order.

Exercise: `max2`

The method `max2` below takes in two integers and returns whichever value is greater.

```
public int max2(int x, int y)
{
   if (x > y)
      return x;
   else
      return y;
}
```

Write an equivalent MIPS subroutine named `max2`, and test that you can call it correctly.

Exercise: `max3`

The method `max3` below takes in three integers and returns whichever value is greatest.

```
public int max3(int x, int y, int z)
{
   return max2(max2(x, y), z);
}
```

Write an equivalent MIPS subroutine named `max3`, and test that you can call it correctly. In implementing `max3`, be sure to call your `max2` subroutine.

Exercise: `fact`

The method `fact` below takes in a non-negative integer *n* and returns *n*! (*n* factorial).

```
public int fact(int n)
{
   if (n == 0)
      return 1;
   else
      return n * fact(n – 1);
}
```

Write an equivalent MIPS subroutine named `fact`, and test that you can call it correctly. Your `fact` subroutine should call itself, just like the recursive Java method shown above.

Next:

- Implement a subroutine `fib` based on the following method.

```
public int fib(int n)
{
   if (n <= 1)
      return n;
   else
      return fib(n – 1) + fib(n – 2);
}
```

- Optionally implement a subroutine `newlistnode` based on the following method.

```
public ListNode newlistnode(int value, ListNode next)
{
   return new ListNode(value, next);
}
```

  Your method will need to allocate 8 bytes of memory, store `value` in the first 4 bytes and `next` in the next 4 bytes, and then return the address of the first byte. Test that you can use this subroutine to create a simple linked list. Use the address `0` to represent `null`.

- Implement a subroutine `sumlist` based on the following method.

```
public int sumlist(ListNode list)
{
   if (list == null)
      return 0;
   else
      return list.getValue() + sumlist(list.getNext());
}
```