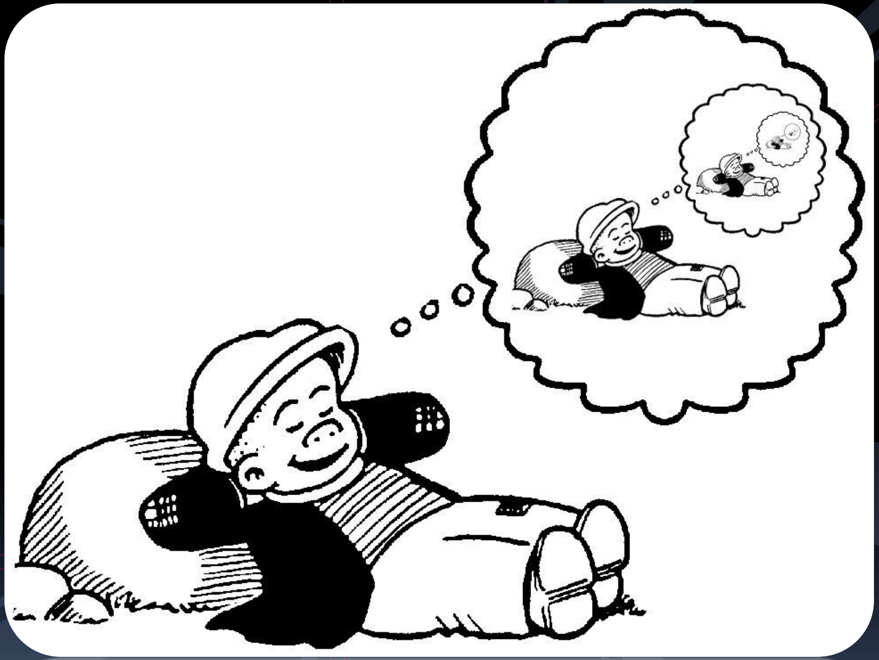
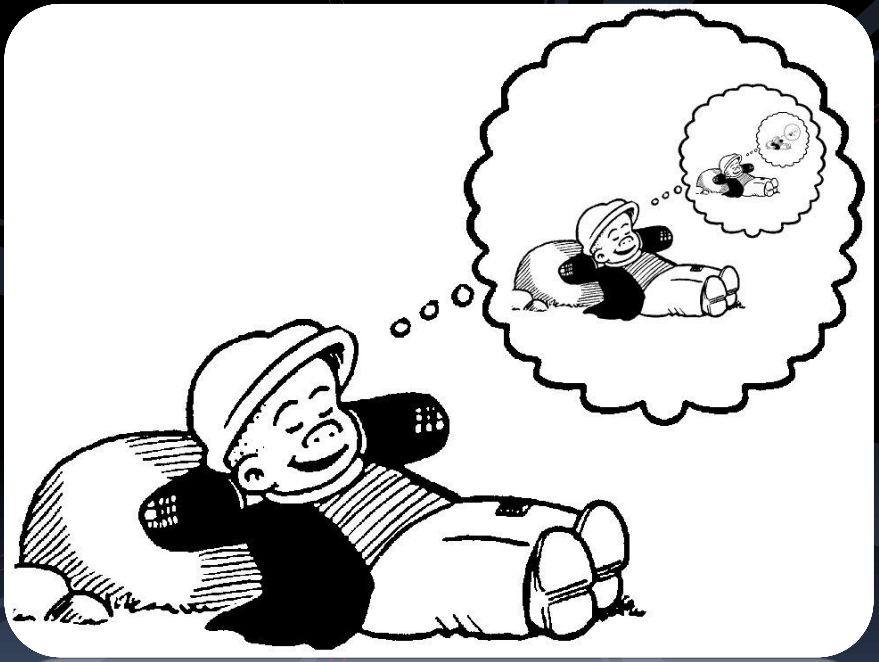
Week 10, part C: Recursion in Assembly



Week 10, part C: Recursion in Assembly



Example: factorial(int n)

Basic pseudocode for recursive factorial:

- Base Case (n == 0)
  - return 1
- Get factorial(n-1)
  - Store result in "temp"
- Multiply temp by n
  - Store in result variable
- Return result

Recursive programs

- How do we handle recursive programs?
  - Still needs base case and recursive step, as with other languages.
  - Main difference: function is both caller and callee
  - So what?
  - Just make sure to preserve $ra and saved registers
    - (As we saw before)

```
int factorial (int x) {
    if (x==0)
        return 1;
    else
        return x*fact(x-1);
}
```

Recursive programs

- <u>Solution</u>: the stack!
  - Before recursive call, store the register values that you use onto the stack, and restore them when you come back to that point.
  - Don't forget to store $ra, or else the program will loop forever!
    - Because it is returning to the wrong place, just after the recursive call.

```
int factorial (int x) {
   if (x==0)
       return 1;
   else
       return x*fact(x-1);
}
```

Factorial solution

```
int fact (int x) {
    if (x==0)
        return 1;
    else
        return x*fact(x-1);
}
```

- Steps to perform:
  - Pop `x` off the stack.
  - Check if `x` is zero:
    - If `x==0`, push `1` onto the stack and return to the calling program.
    - If `x!=0`, push `x-1` onto the stack and call factorial again (i.e. jump to the beginning of the code).
    - After recursive call, pop result off of stack and multiply that value by `x`.
    - Push result onto stack, and return to calling program.

factorial(int n)

- Base Case (n == 0)
  - return 1
- Get factorial(n-1)
  - Store result in "product"
- Multiply product by n
  - Store in "result"
- Return result

n → \$t0

n-1 → \$t1

fact(n-1) → \$t2

factorial(int n)

- Pop n off the stack
  - Store in `$t0`
- If `$t0 == 0`
  - Push 1 onto stack
  - Return to caller
- If `$t0 != 0`

- Base Case (n == 0)
  - return 1
- Get factorial(n-1)
  - Store result in "product"
- Multiply product by n
  - Store in "result"
- Return result

n → $t0

n-1 → $t1

fact(n-1) → $t2

factorial(int n)

- Pop n off the stack
  - Store in `$t0`
- If `$t0 == 0`
  - Push 1 onto stack
  - Return to caller
- If `$t0 != 0`
  - Calculate n-1 in `$t1`
  - Save `$t0` (n) and `$ra` onto stack
  - Push n-1 (`$t1`) onto the stack
  - Call factorial
    - …time passes…

- Base Case (n == 0)
  - return 1
- Get factorial(n-1)
  - Store result in "product"
- Multiply product by n
  - Store in "result"
- Return result

n → $to
n-1 → $t1
fact(n-1) → $t2

`$t0` is caller-saved!

`factorial(int n)`

- Pop n off the stack
  - Store in `$t0`
- If `$t0 == 0`
  - Push 1 onto stack
  - Return to caller
- If `$t0 != 0`
  - Calculate n-1 in `$t1`
  - Save `$t0` (n) and `$ra` onto stack
  - Push n-1 (`$t1`) onto the stack
  - Call factorial
    - …time passes…
  - Pop the result of factorial(n-1) from stack, store in `$t2`
  - Restore `$ra` and `$t0` from stack
  - Multiply factorial(n-1) (stored in `$t2`) and n (in `$t0`)
  - Push result onto stack
  - Return to calling program

`$t0` is caller-saved!

- Base Case (n == 0)
  - return 1
- Get factorial(n-1)
  - Store result in "product"
- Multiply product by n
  - Store in "result"
- Return result

n → $to
n-1 → $t1
fact(n-1) → $t2

Translated recursive program (part 1)

```
main:          add $t3, $zero, 6         #
               addi $sp, $sp, -4         #
               sw $t3, 0($sp)            #
               jal factorial             #
               ...
-------------------------------------------------------------
factorial:     lw $t0, 0($sp)            # pop n off the
               addi $sp, $sp, 4          #   stack
               bne $t0, $zero, rec       # if n is zero?

base:          addi $t0, $zero, 1        # base case:
               addi $sp, $sp, -4         #   push 1 to the
               sw $t0, 0($sp)            #   stack
               jr $ra                    # return to caller

rec:           ...                       # recursive case
```

Translated recursive program (part 2)

```
# (... continuing from part 1)
rec:        addi $t1, $t0, -1          # calculate n-1
            addi $sp, $sp, -4          # save n on the
            sw $t0, 0($sp)             #   stack
            addi $sp, $sp, -4          # save $ra on the
            sw $ra, 0($sp)             #   stack
            addi $sp, $sp, -4          # push n-1 on the
            sw $t1, 0($sp)             #   stack

            jal factorial              # call factorial(n-1)

            # returning from recursive call...
            lw $t2, 0($sp)             # pop result of
            addi $sp, $sp, 4           #   factorial(n-1)
            lw $ra, 0($sp)             # restore saved $ra
            addi $sp, $sp, 4           #   from the stack
            lw $t0, 0($sp)             # restore saved n
            addi $sp, $sp, 4           #   from the stack
```

Translated recursive program (part 2)

```
# (... continuing from part 2)

            mult $t0, $t2                 # compute result
            mflo $t0                      #  n * factorial(n-1)
                                          #  assume no overflow

            addi $sp, $sp, -4             # push result onto
            sw $t0, 0($sp)                #  stack
            jr $ra                        # return to caller
```

The code is available on Quercus.

Highly recommended: download and execute it on MARS.

Factorial Stack View

Now running: main

Initial call to `factorial with x=6`

x = 6

Factorial Stack View

Now running: fact(6)

pop x=6

Factorial Stack View

Now running: fact(6)

save x on the stack

`saved x = 6`

Factorial Stack View

Now running: fact(6)

save $ra in preparation for recursion

$ra for fact(6)
saved x = 6

Factorial Stack View

Now running: fact(6)

push argument for factorial(5)

x = 5
$ra for fact(6)
saved x = 6

factorial(6)

Factorial Stack View

Now running: fact(5)

pop x = 5

factorial(6)

`$ra for fact(6)`
`saved x = 6`

Factorial Stack View

Now running: fact(5)

save $ra in preparation for recursion

`$ra for fact(5)`
`saved x = 5`
`$ra for fact(6)`
`saved x = 6`

factorial(6)

Factorial Stack View

Now running: fact(5)

Factorial Stack View

Now running: fact(4)

factorial(4)	`$ra for fact(4)`
	`saved x = 4`
factorial(5)	`$ra for fact(5)`
	`saved x = 5`
factorial(6)	`$ra for fact(6)`
	`saved x = 6`

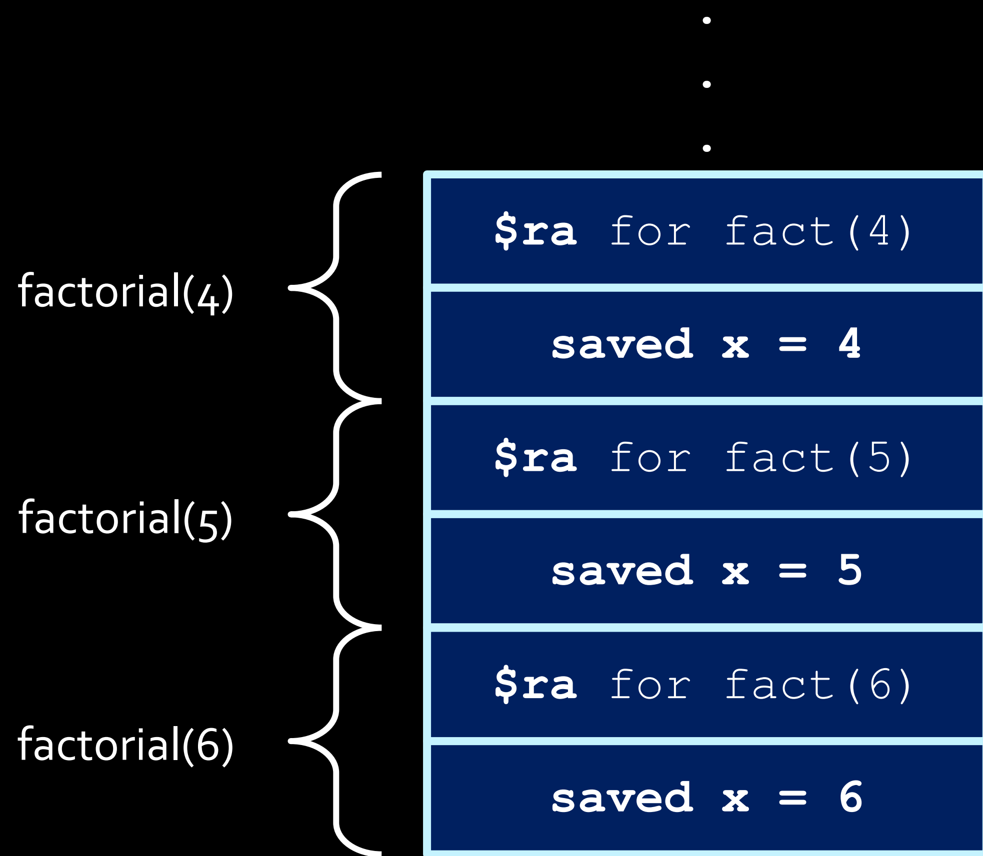
Factorial Stack View

Now running: fact(0)

factorial(0)	x = 0
	. . .
factorial(4)	$ra for fact(4)
	saved x = 4
factorial(5)	$ra for fact(5)
	saved x = 5
factorial(6)	$ra for fact(6)
	saved x = 6

Factorial Stack View

Now running: fact(0)



Factorial Stack View

Now running: fact(0)

return value of fact(0)

1

.
.
.

factorial(4)

`$ra for fact(4)`

`saved x = 4`

factorial(5)

`$ra for fact(5)`

`saved x = 5`

factorial(6)

`$ra for fact(6)`

`saved x = 6`

Factorial Stack View

fact(0) is now returning

return value of fact(0)

	1
factorial(1)	$ra for fact(1)
	saved x = 1
factorial(2)	$ra for fact(2)
	saved x = 2

.
.
.

factorial(6)	$ra for fact(6)
	saved x = 6

Factorial Stack View

Now running: fact(1)

pop fact(0) and saved registers

factorial(2)

$ra for fact(2)
saved x = 2

.
.
.

factorial(6)

$ra for fact(6)
saved x = 6

Factorial Stack View

Now running: fact(1)

push 1 * fact(0) and return

1
$ra for fact(2)
saved x = 2

factorial(2)

. . .

$ra for fact(6)
saved x = 6

factorial(6)

Factorial Stack View

Now running: fact(1)

1
`$ra for fact(2)`
`saved x = 2`

factorial(2)

pop fact(1) and saved values

.
.
.

`$ra for fact(6)`
`saved x = 6`

factorial(6)

Factorial Stack View

Now running: fact(1)

factorial(2)

pop fact(1) and saved values

.
.
.

factorial(6)

$ra for fact(6)
saved x = 6

Factorial Stack View

Now running: fact(2)

push 2 * fact(1) as return value

.
.
.

factorial(6)

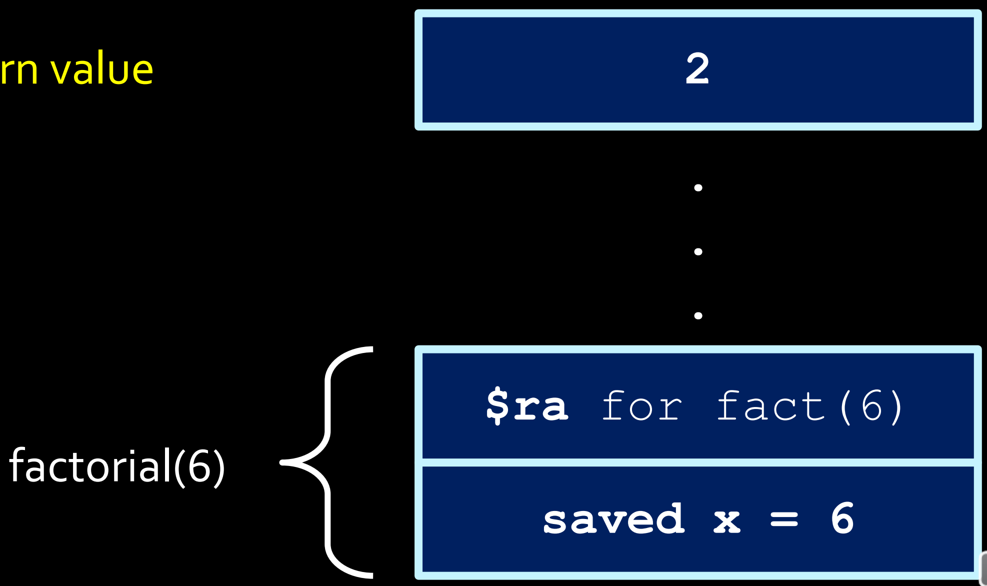
$ra for fact(6)

saved x = 6

Factorial Stack View

Now running: fact(2)

push 2 * fact(1) as return value



Factorial Stack View

Now running: fact(2)

push 2 * fact(1) as return value

2

.
.
.

factorial(6)

$ra for fact(6)

saved x = 6

Factorial Stack View

Now running: fact(5)

return value of fact(5)

factorial(6)

120
$ra for fact(6)
saved x = 6

Factorial Stack View

Now running: fact(6)

push return value 6*fact(5)

Factorial Stack View

Now running: fact(6)

push return value 6*fact(5)

720

Factorial Stack View

Now running: main

main pops result from stack (720)

Reflections on Recursion

- Assembly does not understand recursion.
- Assembly programs are just a linear sequence of assembly instructions.
- Recursion comes from:
  - Jumping to the beginning of the function over and over again and….
  - Using the stack sensibly to store and retrieve remembered values from the stack
- C function signatures help the compiler implement the recursion (what to push/pop)

The Stack is Finite

- You can recurse too much!
  - Maximum stack size limits the number of recursive calls that you can make.
  - Also depends on how much the function is using the stack.
- Exceeding the limit is called a stack overflow.
  - Older systems – overwrite variables, crashes…
  - On later systems, you might get an exception.
- Modern systems can grow the stack dynamically.

Local Variables

- Sometimes you just need local variables
  - You ran out of registers.
  - Or you want a local array or local struct.
  - You are compiling C code and the programmer is using many local variables.

→ In next part!

```
int func(int a, int b) {
    int local_array[256];
    ...
}
```