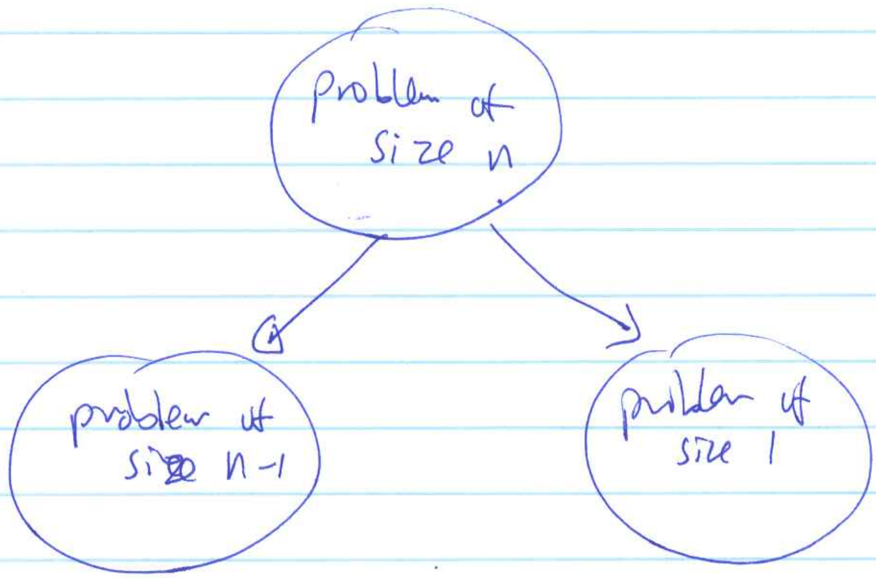
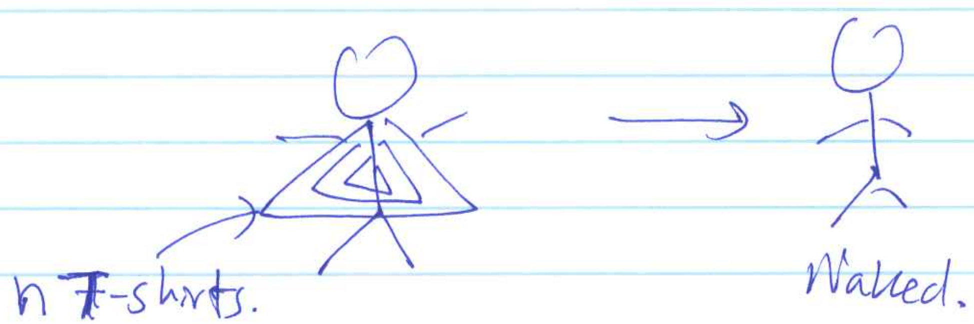
Recursion

- Is a problem solving technique:



Example:



Problem: take off n T-shirts.

take off n-1 T-shirts — take off 1 T-shirt.

(bra).

New 11/1

Recursion

- A recursive function calls itself.

Eg: factorial function:

```
int fac (int n)
  {
      if (n > 0)
          return( n * f(n-1));
      else
          return(1);
  }
```

- Each "invocation" of a recursive subroutine has its own private set of:

  (1) input parameters
  (2) local variables.

  } represent state information!

Schematically, a recursive subroutine in assembler code looks like this:

```
fac:    ...
        ...        ]  <- through out the body
        ...        ]     of the subroutine,
        ...        ]     the input parameters
                         & local variables
                         must have the valid
                         values.
        jsr fac       <- call itself.
        ...        ]
        ...        ]
        ...        ]
        rts
```

Let's see what happens if we pass parameters in registers.

```
main()
  {
    x=fac(4);
    ...
  }
```

```
int fac (int n)
  { if (n > 0)
        return (n * fac(n-1));
    else
        return(1);
```

```
main:   ..
        ..
        move.l #4, D0
        jsr    fac
        move.l D7, X
```

```
fac:          cmp.l  #0, D0
              ble    ElsePart

(save n)      move.l D0, save
              sub.l  #1, D0
              jsr    fac

              move.l save, D0
              muls   D0, D7
              rts

ElsePart:     move.l #1, D7
              rts

save:
```

In: D0
out: D7.

Execution:

	D0	D7	save
move.l #4,D0	4		
jsr fac			
cmp.l #0, D0			
ble ElsePart (no br)			
move.l D0, save			4
sub.l #1, D0	3		
jsr fac (compute fac(3), when it returns, we can multiply result in D7 with 4 to get 24 !!!).			
cmp.l #0, D0			
ble ElsePart (no br)			
move.l D0, save			3 !!!
sub.l #1, D0	2		
jsr fac (compute fac(2), when it returns, we can multiply result in D7 with 3 to get 6.)			

	D0	D7	Save	
cmp.l #0, D0				
ble ElsePart (no br)				
move.l D0, Save			2	
sub.l #1, D0	1			
jsr fac				– Compute fac(1).
Cmp.l #1, D0				
ble ElsePart (BR!!!)				
move.l #1, D7		1		↑ fac(1) = 1 Correct.
rts				back to jsr fac.
move.l save, D0	2			
muls D0, D7		2		↑ fac(2) = 2 correct.
rts				back to jsr fac.

		D0	D7	Save
move.l	save, D0	2		
muls	D0, D7		4	
			↑ fac(3) = 4 ???	
rts				
move.l	save, D0	2		
muls	D0, D7 (2) (4)		8	
			↑ fac(4) = 8 ???	
rts	back to main.			

What caused the error?

first call:

fac : — D∅ = 4.

—

—

—

—

—

—

JSR fac

— ← When JSR fac returns save was changed to 2!!! so dropout is no longer valid!!!

D∅ = 4

Save: 4.

How do we solve this problem? (you can't cure a patient before finding the right diagnosis.)

- First: understand the cause of the problem:

Recursion:

each invocation of the fac subroutine has

(1) its own input parameters (can be different for diff. invocations)

(2) its own local variables.

Problem:

- [~~The~~ recursive function paused while it was active. While it was paused, ~~the~~ some ~~data~~ of its data in reg's or memory got corrupted.]

Passing parameter in registers:

there is one copy of a registers
- diff. invocations of recursive subroutines cannot share same register for input.

Local variables in memory:

The same location in memory can not be used by diff. invocations of recur. subroutines to hold diff. ~~treat~~ values.

- The solution is to use a structure that grows ~~and~~ shrinks IN THE SAME WAY as subroutine call & return.

This structure is of course a stack.

→ pass parameters on stack
allocate local variables on stack

↑ in order to make recursion work!

- We must deal with 2 things:

  (1) How to pass parameters on stack.

  (2) How to "allocate" local variables on stack.

- Before doing these, we must learn ~~a more of~~ instructions to push & pop the system stack.

New addressing modes:

- Indirect with post increment

(1) (An)+ (seen before where we traverse an array).

effect: use (An) as effective addr. then increment An to make it point to next "item".

eg:

```
move.l (A0)+, D0
```

is equivalent to the following 2 instructions:

```
move.l (A0), D0
adda.l #4, A0.
```

Summary

move.l (A0)+, D0	move.l (A0), D0 adda.l #4, A0
move.w (A0)+, D0	move.w (A0), D0 adda.l #2, A0
move.b (A0)+, D0	move.b (A0), D0 adda.l #1, A0

(2) indirect with pre-decrement

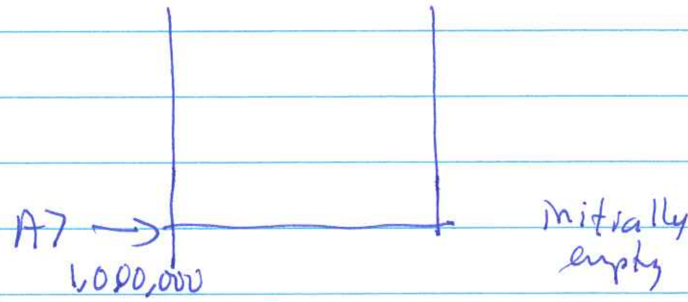
−(An)

effect: First decrement An to make it point to the "previous item". Then use (An) as effective address.

eg:

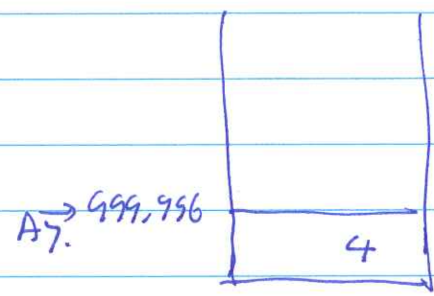
move.l −(A0),D0	~~move.l (~~ suba.l #4, A0 move.l (A0), D0
move.w −(A0), D0	suba.l #2, A0 move.l (A0), D0
move.b −(A0), D0	suba.l #1, A0 move.l (A0), D0

How do we push & pop thing on the system stack.

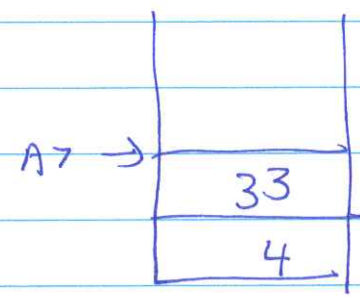


~~move.l D0,~~

move.l #4, -(A7)
(push 4 on syst. stack)



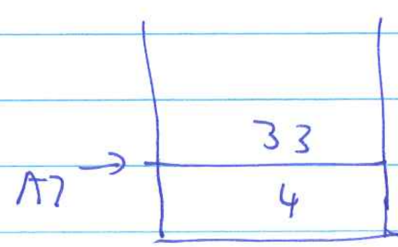
move.l #33, -(A7)
(push 33 on stack)



in general: move.l x, -(A7) pushes x on sys. stack.

move.l (A7)+, DØ
pops ~~first long~~ long in DØ

DØ = 33.



Recursion: the requirements

Recall:

(1) each invocation of a recursive subroutine has its own set of input parameters

(2) Each invocation of a recursive subroutine has its own set of local variables.

Recursion: calling sequence

Recursive subroutine calls are also sequenced as "First In Last Out".

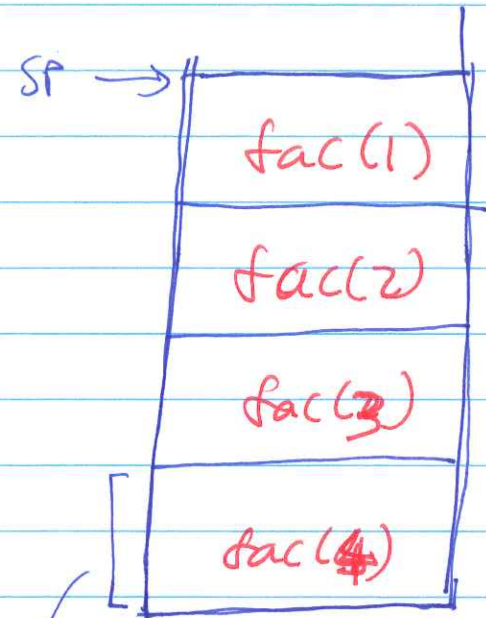
Conclusion.

→ A stack is the proper structure to support passing of parameters & allocating local variables.

Recursion : the global picture

fac :

jsr fac



block for one subroutine call "Activation Record" or a "frame".

Key to making recursion work :

(1) "block" of data structure to store parameters & local variables on the stack are identical in <u>structure</u>. (Activation record)

(2) The subroutine ALWAYS use the activation record on the TOP of the stack.
(only one subroutine is active).

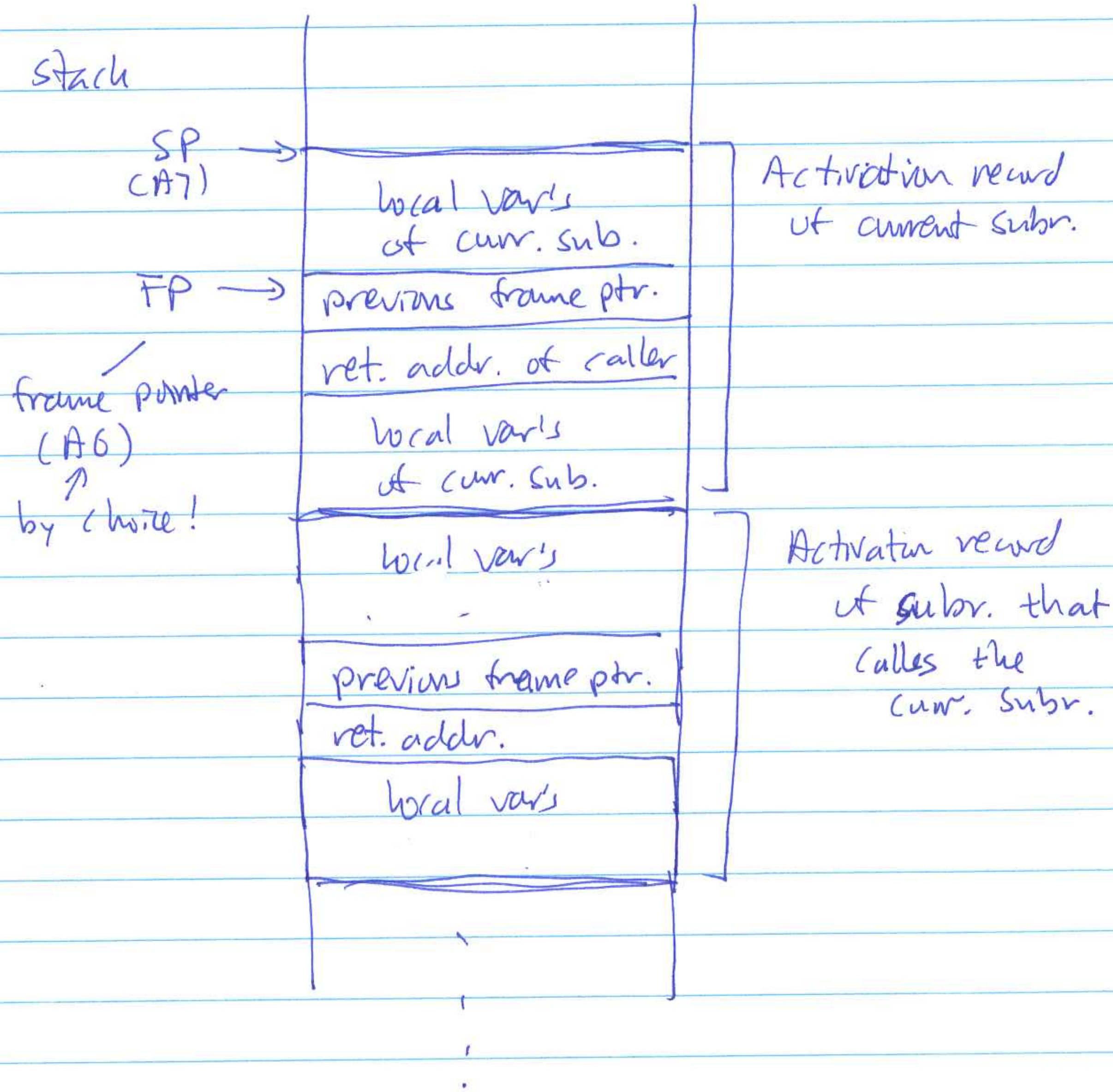
Activation records are ~~created~~ (push on the stack) when a recursive subroutine is called.

Activation records are ~~destroyed~~ popped off the stack when a recursive subroutine exits.

→ the stack grows & shrinks with recursion calls & returns !!!

Format Activation Record (Frame format)

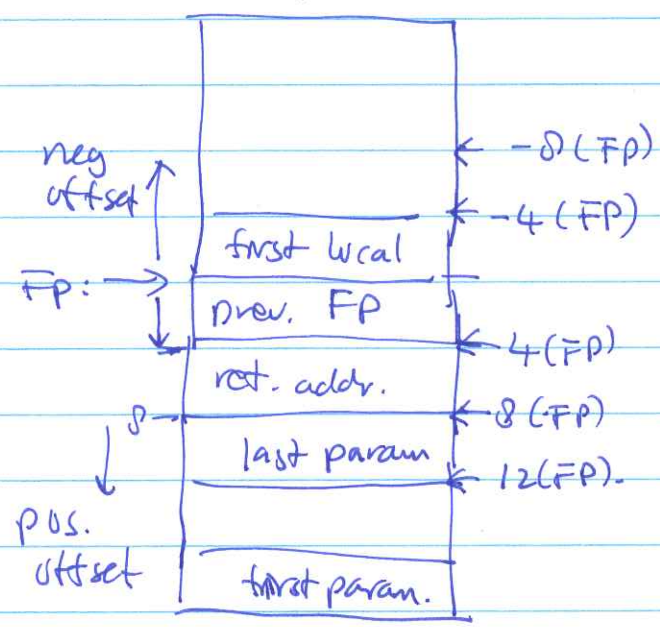
The most often used format is as follows:



Notes:

(1) A special pointer (address reg A6) is used to point to the division of the frame. This pointer is called "Frame pointer". (FP)

Usage:



- neg. offset from FP are local variables.
- pos. offset (≥ 8) from FP are the input parameters.

(2) Special note: Frame contains a thing for storing "prev. FP".

It is necessary to save caller's FP because callee will destroy (over write) the FP register.

- There is a very good reason why the ~~parameters~~ parameters are under the return address:

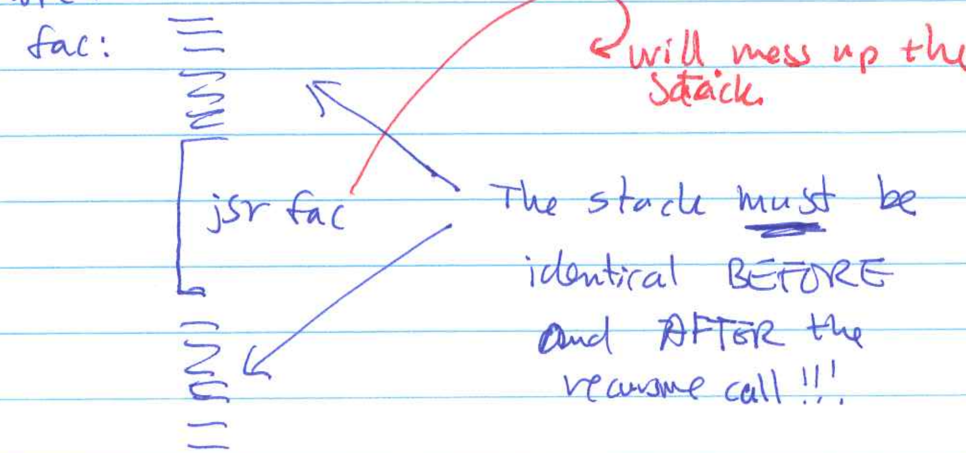
  caller pushes them before executing the jsr-instruction!

- Note:

  the activation record must be built and you will see assembler instructions that manipulate the system stack.

  Also: BOTH the caller and the callee pitch in to build the activation record of the callee !!!

- Special note:



Global picture :

main :

⋮

pass param on stack

jsr fac

clean up parameters

⋮

fac : complete frame intro

look at top of stack for parameters & local var's.

pass param on stack to itself

jsr fac

clean up parameters

look at top of stack for param's & local var's.

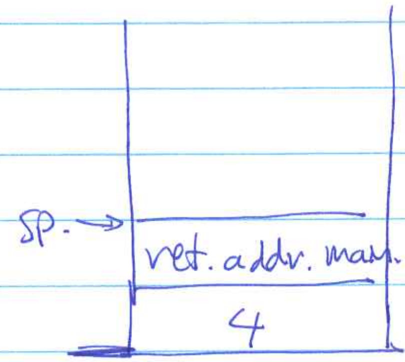
clean up frame made at intro.

exit

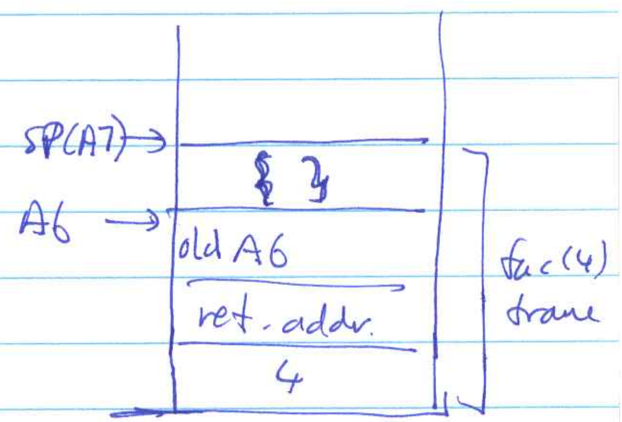
step-by-step:

stack

when main calls fac:
with param = 4

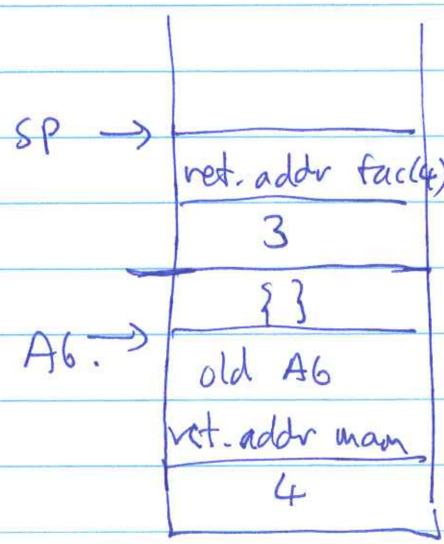


fac first completes the frame:

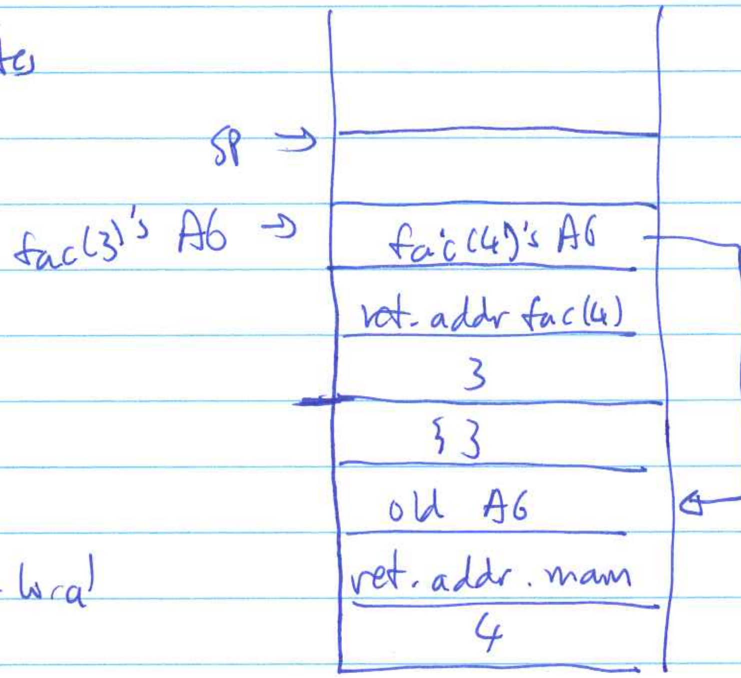


fac uses A6 to access param's & local var's

when fac(4) calls fac with param 3:



When fac(3) completes
the frame:



fac uses A6 to
access param's & local
var's
— does not
overwrite fac(4)'s copy!

When fac(3) returns, the stack MUST be
returned back to the picture before fac(3)
was called!

Recursion: a more detailed picture

main:

⋮

[push parameters on stack
jsr fac
pop away parameters]

⋮

make sure the stack is the same before AND after the recursion call!!.

You need to clean them up, otherwise the stack is screw up.

(1)

fac:

prelude [complete frame:
(1) push A6
(2) A6 ← A7
(3) allocate local variables.]

⋮

use A6 to access parameters & local var's on stack

make sure stack unchange (same) before & after call!!!

[← push param. on stack
jsr fac
← pop away useless param's]

⋮

postlude [remove portion of frame allocated in prelude
(1) A7 ← A6
(2) pop A6.]
rts

(2)

When jsr fac is executed, the stack is like this:

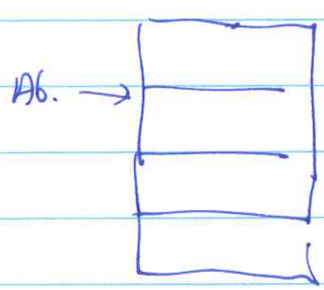
frame is incomplete.

Finally: a recursive subroutine

```
main()
{ x = fac(4);
}
```

result in D7

```
main:
      ⋮
      move.l  #4, -(A7)
      jsr     fac
      adda.l  #4, A7
      move.l  D7, X
      ⋮
```




```
int fac (int n)
{  if (n > 1)
      return (n * fac(n-1))
   else
      return (1)
}
```

Prelude

```
fac:  move.l  A6, -(A7)
      move.l  A7, A6
      (suba.l #0, A7)
      — ready

      move.l  8(A6), D0
      cmp.l   #1, D0
      ble     Else

Call fac(n-1)
      move.l  8(A6), D0
      sub.l   #1, D0
      move.l  D0, -(A7)
      jsr     fac
      adda.l  #4, A7

n*fac(n-1)
      move.l  8(A6), D0
      muls    D0, D7
```

postlude

```
move.l  A6, A7
move.l  (A7)+, A6
rts
```

Else:

```
move.l  #1, D7
```

postlude

```
move.l  A6, A7
move.l  (A7)+, A6

rts
```

Recursion: a receipe

Calling a recursive subroutine:

```
move[.l]    param1 , -(A7)
move[.l]    parame2 , -(A7)
   ⋮
jsr         sub

adda.l      # n-bytes pushed , A7
```

Structure of recursive subroutine:

```
sub:  prelude [  move.l   A6, -(A7)
              [  move.l   A7, A6
              [  suba.l   # n-bytes localvars, A7

      } body subroutine.
           8(A6)     first parameter.
                     and so on
          -4(A6)     first local var.
                     (if long)
                     and so on
```

Before you return to caller with rts,
you must restore stack built in prelude:

postlude

```
move.l  A7, A6
move.l  (A7)+, A6

rts
```

Another example of recursion: fibonacci numbers.

$$f_n = f_{n-1} + f_{n-2}.$$

$$f_1 = 1$$
$$f_0 = 1.$$

So.

$$f_0 = 1$$
$$f_1 = 1$$
$$f_2 = f_1 + f_0 = 1 + 1 = 2$$

$$f_3 = f_2 + f_1 = 2 + 1 = 3$$

$$f_4 = f_3 + f_2 = 3 + 2 = 5.$$

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