

Stacks and Procedures

Lets see, before returning from break. I'd better look over my 6.004 notes... but I'll need to find my backpack first... that means I'll need to find the car... meaning, I'll need to remember where I parked it... maybe it would help if I could remember where I was last night... um, I forget, what was I going to do...



Handouts: Lecture Slides

Procedure Linkage: First Try

```
int fact(int n)
{
    if (n>0)
        return n*fact(n-1);
    else
        return 1;
}
fact(4);
```

Proposed convention:

- pass arg in R1
- pass return addr in R28
- return result in R1
- questions:
 - nargs > 1?
 - preserve regs?

```
fact:
    CMPLC(r1,0,r0)
    BT(r0,else)
    MOVE(r1,r2) | save n
    SUBC(r2,1,r1)
    BR(fact,r28)
    MUL(r1,r2,r1)
    BR(rtn)

else: CMOVE(1,r1)
rtn:  JMP(r28,r31)

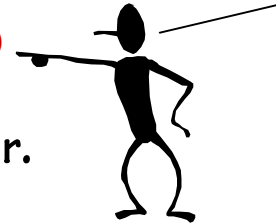
    CMOVE(4,r1)
    BR(fact,r28)
    HALT()
```

OOPS!

A Procedure's Storage Needs

Basic Overhead for Procedures/Functions:

- Arguments
`f(x, y, z)` or worse... `sin(a+b)`
- Return Address back to caller
- Results to be passed back to caller.



In C it's the caller's job to evaluate its arguments as expressions, and pass their resulting values to the callee... Thus, a variable name is just a simple case of an expression.

Temporary Storage:

intermediate results during expression evaluation.

`(a+b) * (c+d)`

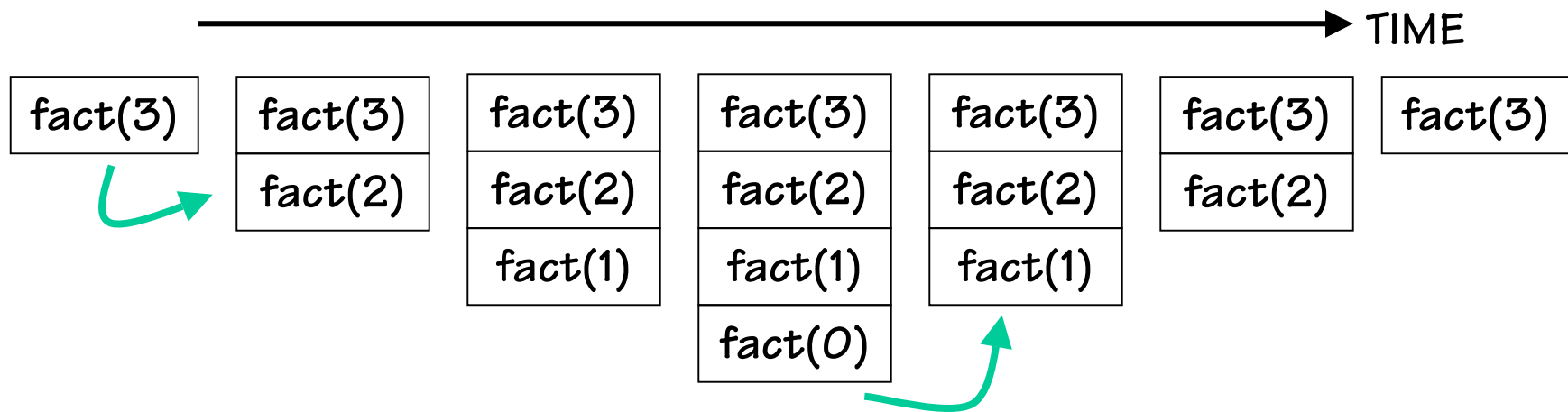
Local variables:

```
...  
{  
    int x, y;  
    ... x ... y ...;  
}
```

Each of these is specific to a particular *activation* of a procedure; collectively, they may be viewed as the procedure's *activation record*.

Lives of Activation Records

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



A procedure call creates a new activation record. Caller's record is preserved because we'll need it when call finally returns.

Return to previous activation record when procedure finishes, permanently discarding activation record created by call we are returning from.

We need a STACK!

What we need is a SCRATCH memory for holding temporary variables. We'd like for this memory to grow and shrink as needed. And, we'd like it to have an easy management policy.

One possibility is a

STACK

A last-in-first-out (LIFO) data structure.



Some interesting properties of stacks:

Little overhead. Only the top is directly visible, the so-called “top-of-stack”

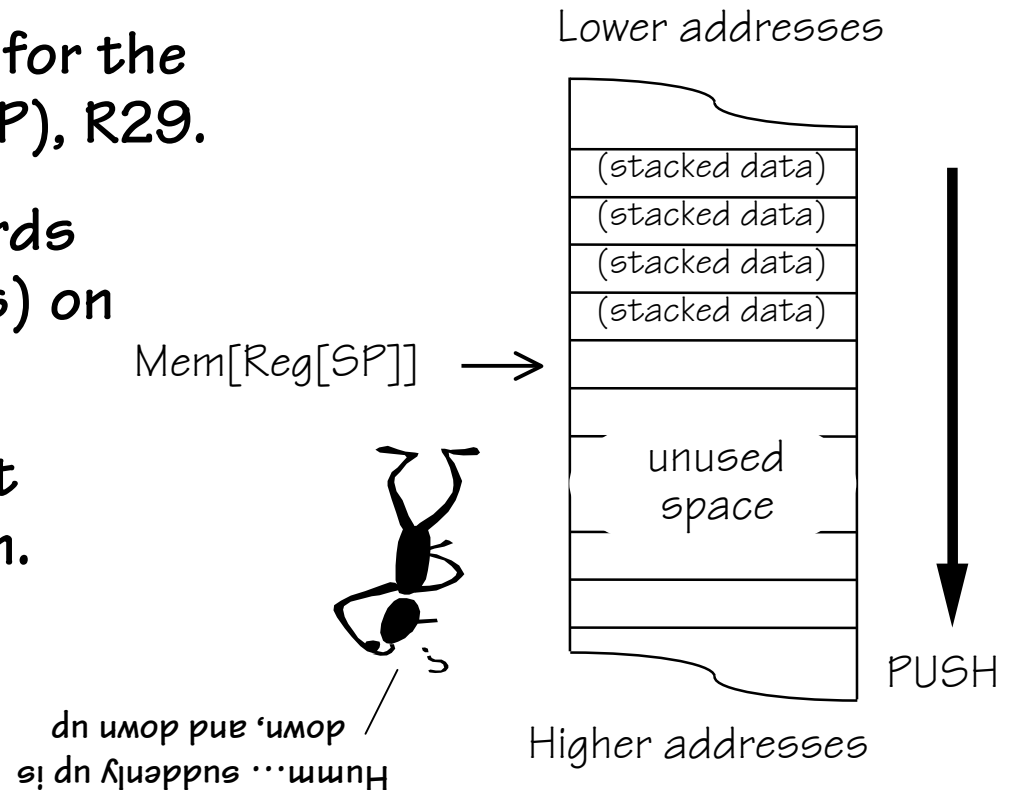
We can add things by PUSHING a new value on top.

We can remove things by POPING off the top value.

Stack Implementation

CONVENTIONS:

- Waste a register for the Stack Pointer (SP), R29.
- Builds UP (towards higher addresses) on push
- SP points to first **UNUSED** location.
- Allocated a lot of memory well away from our program and its data



Other possible implementations include stacks that grow “down”, SP points to top of stack, etc.

Stack Management Macros

PUSH (RX) : push Reg[x] onto stack

$\text{Reg}[\text{SP}] = \text{Reg}[\text{SP}] + 4;$

$\text{Mem}[\text{Reg}[\text{SP}]-4] = \text{Reg}[\text{x}]$

```
ADDC(R29, 4, R29)
ST(RX,-4,R29)
```

POP (RX) : pop the value on the top of the stack into Reg[x]

$\text{Reg}[\text{x}] = \text{Mem}[\text{Reg}[\text{SP}]-4]$

$\text{Reg}[\text{SP}] = \text{Reg}[\text{SP}] - 4;$

```
LD(R29, -4, RX)
ADDC(R29,-4,R29)
```

Safe?



ALLOCATE (k) : reserve k WORDS of stack

$\text{Reg}[\text{SP}] = \text{Reg}[\text{SP}] + 4*k$

```
ADDC(R29,4*k,R29)
```

DEALLOCATE (k) : release k WORDS of stack

$\text{Reg}[\text{SP}] = \text{Reg}[\text{SP}] - 4*k$

```
SUBC(R29,4*k,R29)
```

Fun with Stacks

We can squirrel away variables for latter. For instance, the following code fragment can be inserted anywhere within a program.

```
suspense:
|
| Argh!!! I'm out of registers Scotty!!
|
PUSH(R0)          | Frees up R0
PUSH(R1)          | Frees up R1
LD(R31,dilithum_xtals, R0)
LD(R31,seconds_til_explosion, R1)
SUBC(R1, 1, R1)
BNE(R1, suspense, R31)
ST(R0, warp_engines,R31)
POP(R1)           | Restores R1
POP(R0)           | Restores R0
```

Data is popped off the stack in the opposite order that it is pushed on



AND Stacks can also be used to solve other problems...

Solving Procedure Linkage “Problems”

In case you forgot, a reminder of our problems:

- 1) We need a way to pass arguments into procedures
- 2) Procedures need their own LOCAL variables
- 3) Procedures need to call other procedures
- 4) Procedures might call themselves (Recursion)

BUT FIRST, WE’LL WASTE SOME MORE REGISTERS:

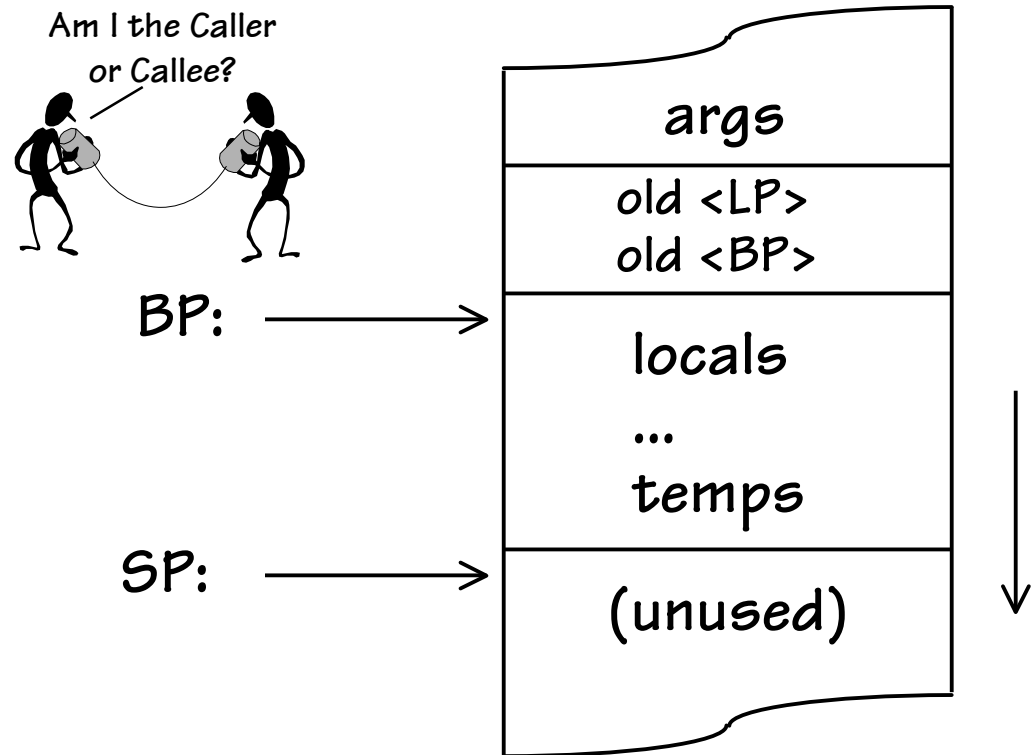
- r27 = **BP**. Base ptr, points into stack at the local variables of callee
- r28 = **LP**. Linkage ptr, return address to caller
- r29 = **SP**. Stack ptr, points to 1st unused word

Then we can define a STACK FRAME
(aka the procedure’s Activation Record):

Stack frame overview

The CALLEE will use the stack for all of the following storage needs:

- 1) saving the RETURN ADDRESS back to the caller
- 2) saving the CALLER's base ptr
- 3) Creating its own local/temp variables



In theory it's possible to use SP to access stack frame, but offsets will change due to PUSHs and POPs. For convenience we use BP so we can use constant offsets to find, e.g., the first argument.

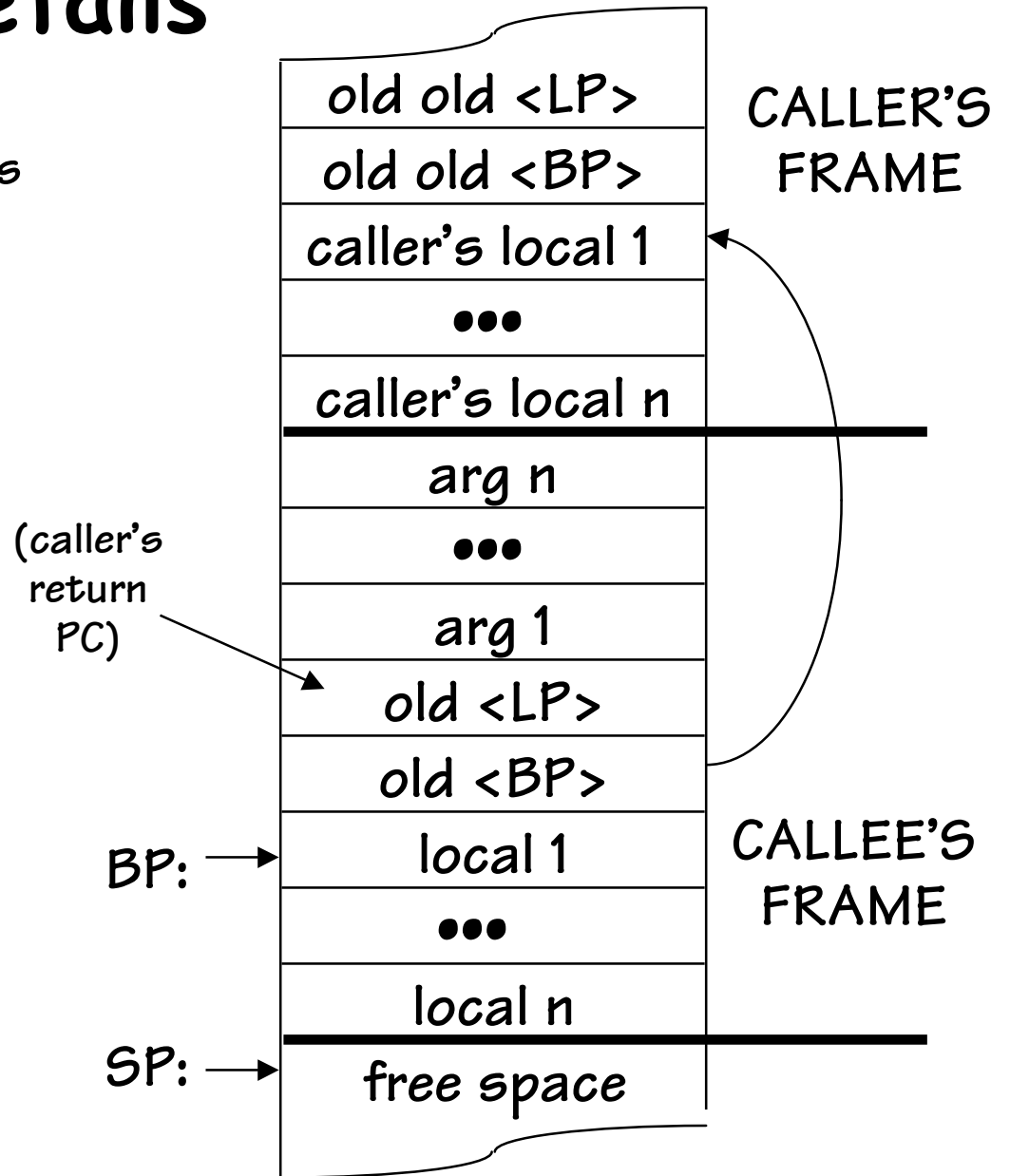
Stack Frame Details

The CALLER passes arguments to the CALLEE on the stack in REVERSE order

F(1,2,3,4) is translated to:

```
ADDC (R31, 4, R0)
PUSH (R0)
ADDC (R31, 3, R0)
PUSH (R0)
ADDC (R31, 2, R0)
PUSH (R0)
ADDC (R31, 1, R0)
PUSH (R0)
BEQ (R31, F, LP)
```

QUESTION: Why push args in REVERSE order???



Order of Arguments

Why push args onto the stack in *reverse order*?

1) It allows the BP to serve double duties when accessing the local frame

To access i^{th} local variable ($i \geq 1$)

LD (BP, (i-1)*4, rx)

or

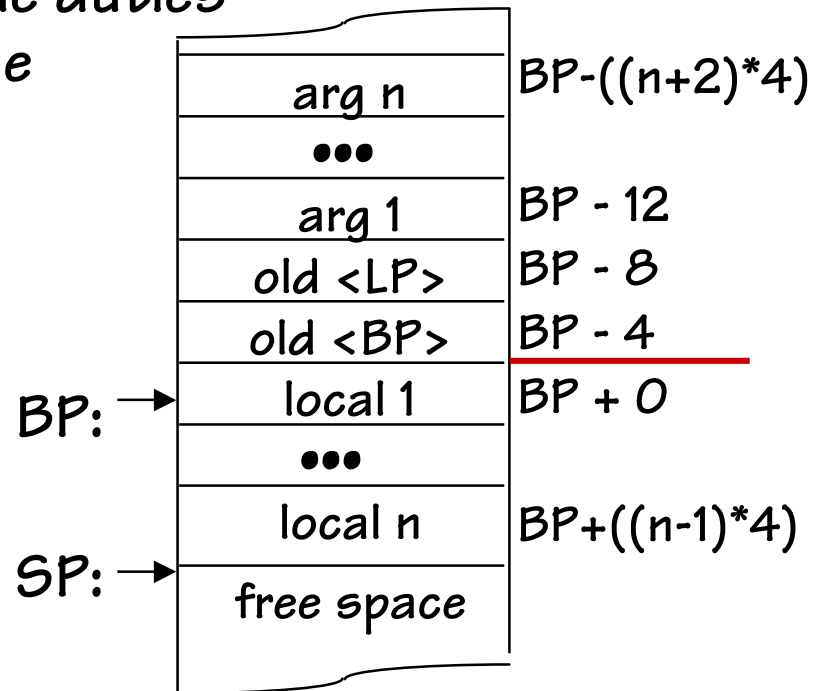
ST (rx, (i-1)*4, BP)

To access j^{th} argument ($j \geq 1$):

LD (BP, -4*(j+2), rx)

or

ST (rx, -4*(j+2), BP)



2) The CALLEE does not NEED to know how many arguments were passed to it!

Procedure Linkage: The Contract

The CALLER will:

- Push args onto stack, in reverse order.
- Branch to callee, putting return address into LP.
- Remove args from stack on return.

The CALLEE will:

- Perform promised computation, leaving result in R0.
- Branch to return address.
- Leave stacked data intact, including stacked args.
- Leave regs (except R0) unchanged.

Procedure Linkage: The Fine Print

Calling Sequence

```
PUSH(argn)           | push args, last arg first
...
PUSH(arg1)
BEQ(R31,f, LP)        | Call f.
DEALLOCATE(n)         | Clean up!
...                   | (f's return value in r0)
```

Entry Sequence

```
f: PUSH(LP)           | Save LP and BP
    PUSH(BP)          | in case we make new calls.
    MOVE(SP,BP)       | set BP=frame base
    ALLOCATE(nlocals) | allocate locals
    (push other regs) | preserve any regs used
```

Return Sequence

```
(pop other regs)      | restore regs
MOVE(val, R0)         | set return value
MOVE(BP, SP)          | strip locals, etc
POP(BP)               | restore CALLER's linkage
POP(LP)               | (the return address)
JMP(LP, R31)          | return.
```

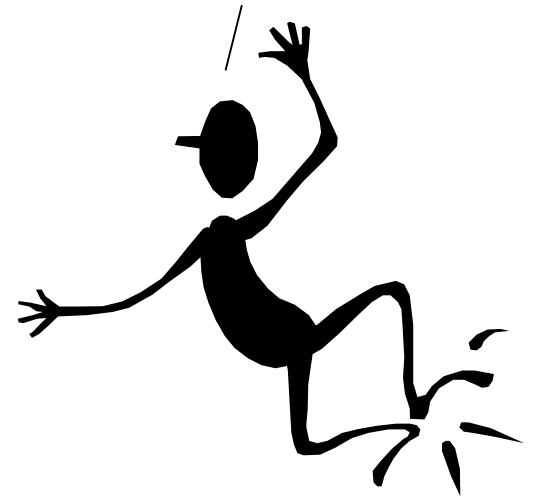
Where's the
Deallocate?



Our favorite subroutine...

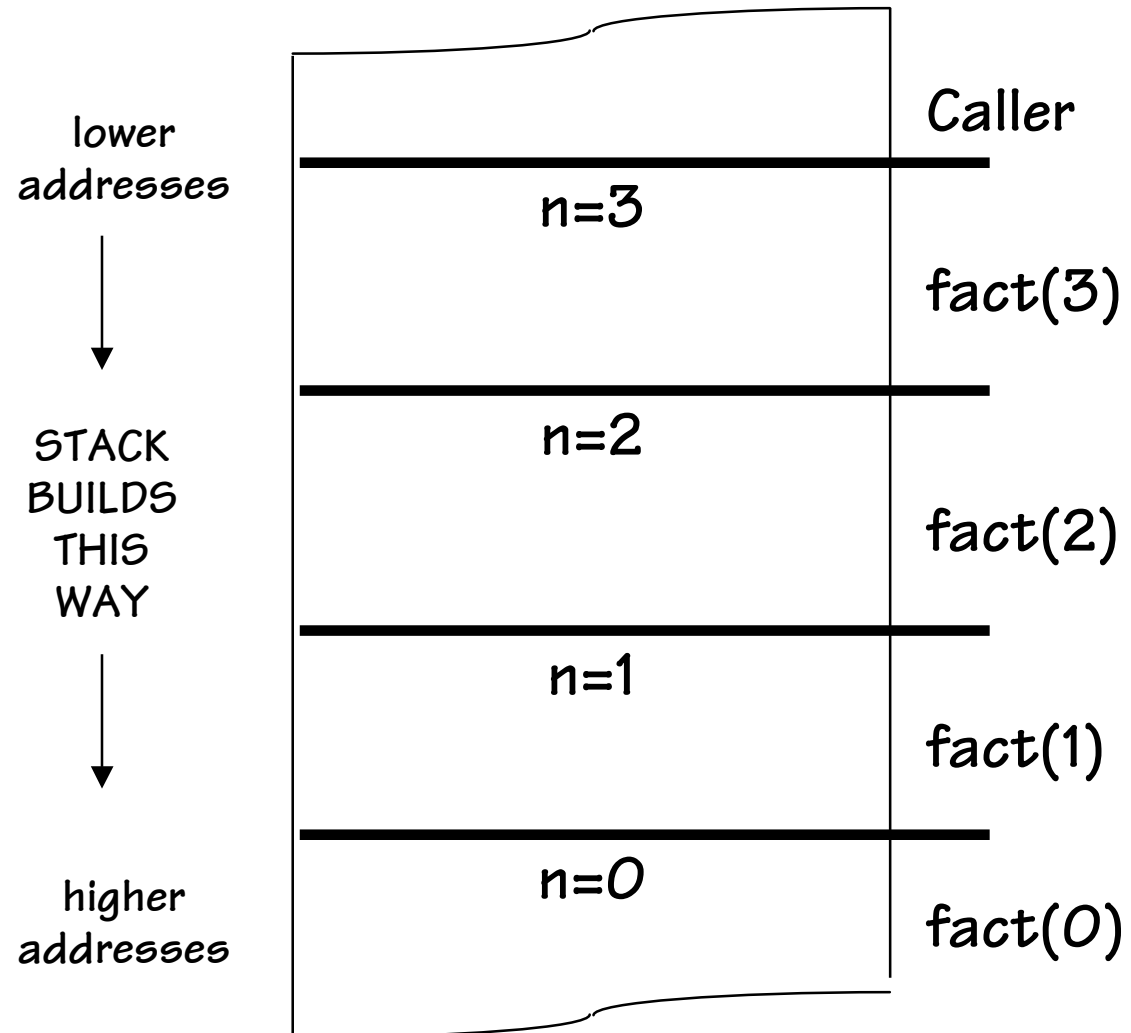
<code>fact:</code>	<code>PUSH(LP)</code>	<code> save linkages</code>	<code>int fact(int n)</code>
	<code>PUSH(BP)</code>		<code>{</code>
	<code>MOVE(SP,BP)</code>	<code> new frame base</code>	<code> if (n == 0)</code>
	<code>PUSH(r1)</code>	<code> preserve regs</code>	<code> return n*fact(n-1);</code>
	<code>LD(BP,-12,r1)</code>	<code> r1 ← n</code>	<code> else</code>
	<code>BNE(r1,big)</code>	<code> if (n == 0)</code>	<code> return 1;</code>
	<code>ADDC(r31,1,r0)</code>	<code> else return 1;</code>	<code>}</code>
	<code>BR(rtn)</code>		
<code>big:</code>	<code>SUBC(r1,1,r1)</code>	<code> r1 ← (n-1)</code>	
	<code>PUSH(r1)</code>	<code> push arg1</code>	
	<code>BR(fact,LP)</code>	<code> fact(n-1)</code>	
	<code>DEALLOCATE(1)</code>	<code> pop arg1</code>	
	<code>LD(BP,-12,r1)</code>	<code> r0 ← n</code>	
	<code>MUL(r1,r0,r0)</code>	<code> r0 ← n*fact(n-1)</code>	
<code>rtn:</code>	<code>POP(r1)</code>	<code> restore regs</code>	
	<code>MOVE(BP,SP)</code>	<code> Why?</code>	
	<code>POP(BP)</code>	<code> restore links</code>	
	<code>POP(LP)</code>		
	<code>JMP(LP,R31)</code>	<code> return.</code>	

Finally, Factorial works!
Now are we done?



This Scheme Supports Recursion

fact(3) ...



Man vs. Machine

Here's a C program which was fed to the C compiler*.
Can you generate code as good as it did?

```
int ack(int i, int j)
{
    if (i == 0) return 2*j;
    if (j == 0) return i+1;
    return ack(i-1, ack(i, j-1));
}
```

* GCC Port courtesy of Cotton Seed & Pat LoPresti;
available on Athena

```
Athena% attach 6.004
```

```
Athena% gcc-beta -S -O2 file.c
```

Tough Problems

1. NON-LOCAL variable access, particularly in nested procedure definitions.

"FUNarg" problem of LISP.

Conventional solution: "static links" in stack frames, pointing to frames of statically enclosing blocks. This allows a run-time discipline which correctly accesses variables in enclosing blocks.

ANALOG: LISP Environments, closures.

[Optional reading: Ward & Halstead section 14.8, p. 400]

(C avoids this problem by outlawing nested procedure declarations!)

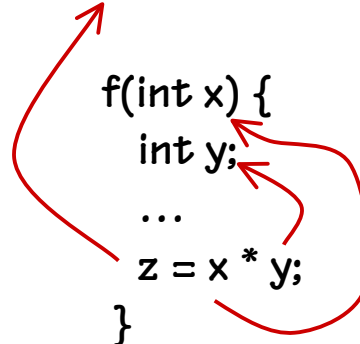
2. "Dangling References" - - -

```
int x, y, z;
```

```
g(int x) {  
  int z;
```

```
  f(int x) {  
    int y;  
    ...  
    z = x * y;  
  }
```

```
  ...  
  f(4);  
}
```



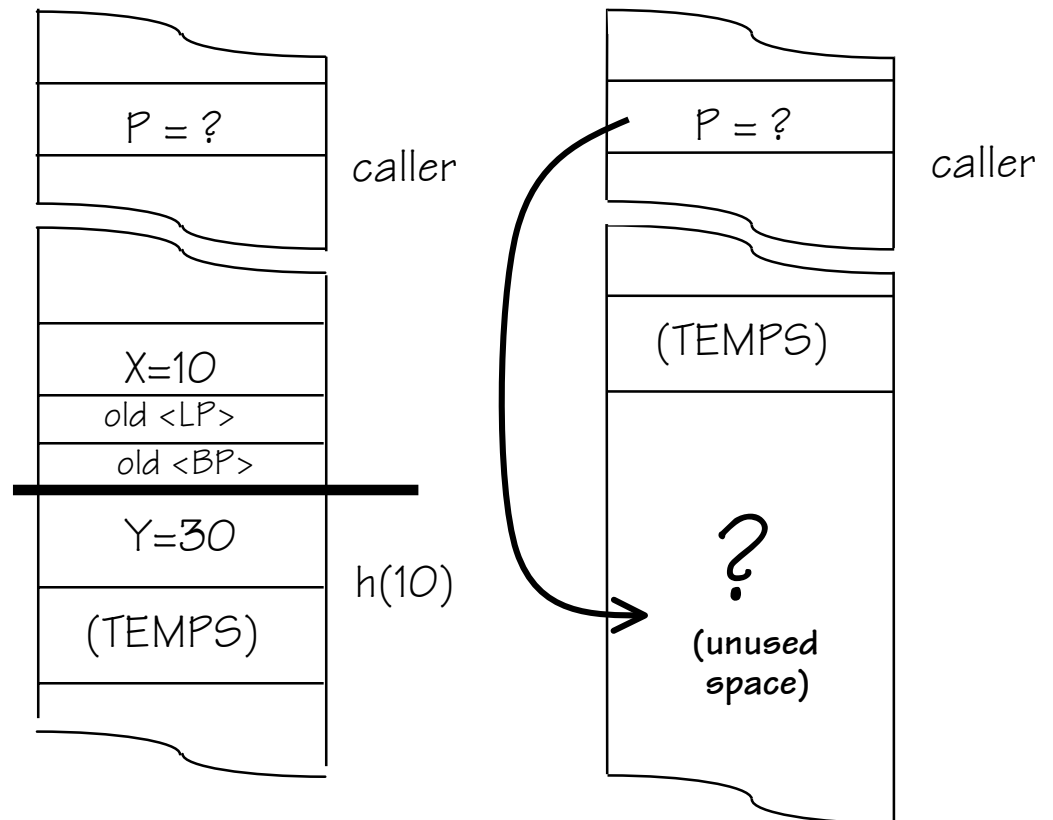
Dangling References

```
int *p; /* a pointer */
```

```
int h(x)  
{  
    int y = x*3;  
    p = &y;  
    return 37;  
}
```

```
h(10);  
print(*p);
```

What do we expect
to be printed?



The Word on Dangling References

Java & PASCAL: kiddie scissors only.

No "ADDRESS OF" operator: language restrictions *forbid* constructs which could lead to dangling references.

C and C++: real tools, real dangers.

"You get what you deserve".

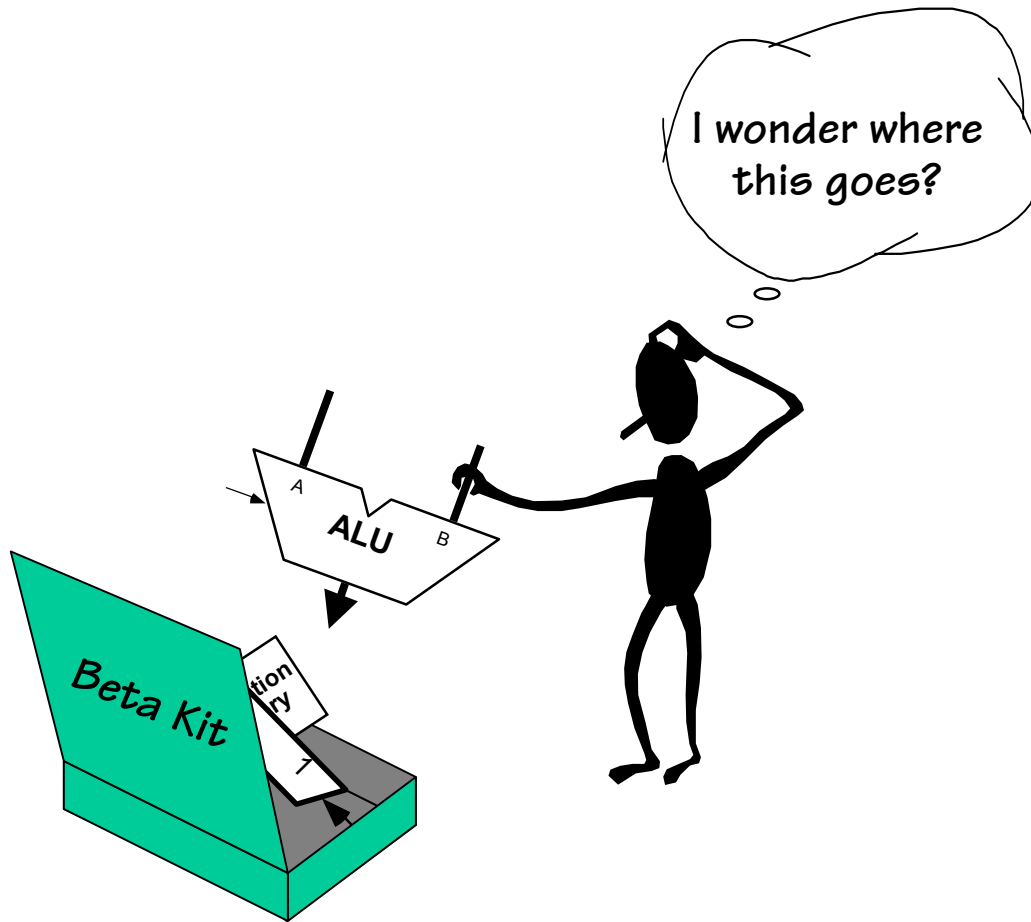
SCHEME/LISP: throw cycles at it.

Activation records allocated from a HEAP, reclaimed transparently by garbage collector (at considerable cost).

"You get what you pay for"

Of course, there's a stack hiding there somewhere...

Next Time: Building a Beta



```
ack:    PUSH (LP)
        PUSH (BP)
        MOVE (SP, BP)
        PUSH (R1)
        PUSH (R2)
        LD (BP, -12, R2)
        LD (BP, -16, R0)
_36:    BNE (R2, _34)
        SHLC (R0, 1, R0)
        BR (_37)
_34:    BEQ (R0, _35)
        SUBC (R2, 1, R1)
        SUBC (R0, 1, R0)
        PUSH (R0)
        PUSH (R2)
        BR (ack, LP)
        MOVE (R1, R2)
        SUBC (SP, 8, SP)
        BR (_36)
_35:    ADDC (R2, 1, R0)
_37:    POP (R2)
        POP (R1)
        POP (BP)
        POP (LP)
        JMP (LP)
```