Implementation of Procedures

– lecture 6 –

Procedure Call Protocol

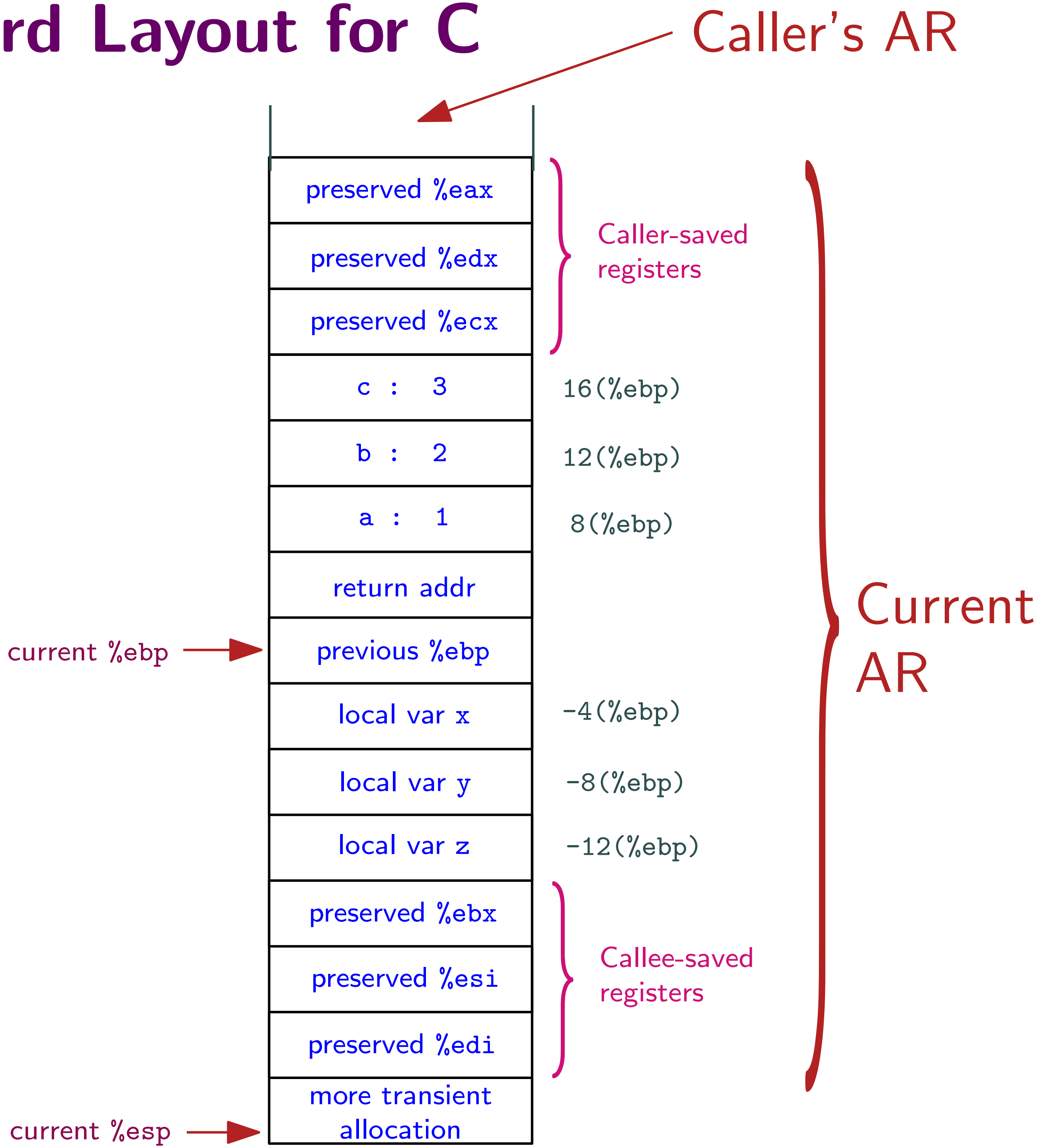
- Defined by processor manufacturer for a specific language.
  - Part of the ABI (Application Binary Interface), which also defines interaction of a program with the OS.
  - Two main conventions: C and Pascal (older).
  - We use the C convention; Pascal convention discussed briefly.

- Activation Record: all the info for executing one instance of the procedure
  - Created on the stack, to allow recursion; callees will have their AR on top of callers'
  - Referenced by a dedicated register, generically called frame pointer; %ebp for x86
  - The stack register separates "allocated" from "unallocated" memory,
    - may vary throughout the execution of the procedure, while %ebp will stay constant

- AR role
  - Stores bindings of actual to formal arguments; arguments referenced as `8(%ebp)`, `12(%ebp)`, `16(%ebp)`, etc
  - Stores the return address to the caller, and the previous value of the frame pointer.
  - Stores all the variables that are local to the procedure instance.

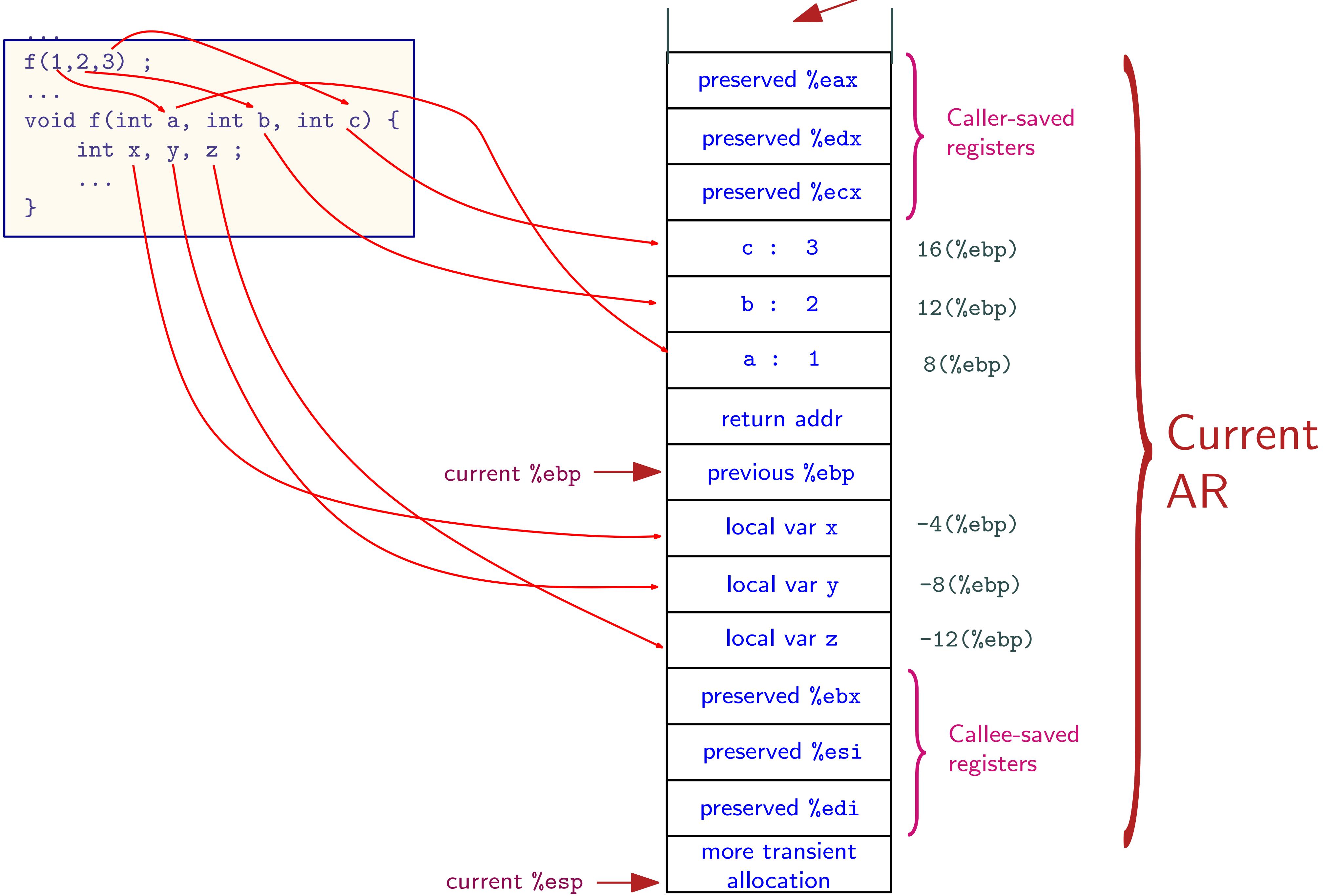
Activation Record Layout for C

```
...
f(1,2,3) ;
...
void f(int a, int b, int c) {
    int x, y, z ;
    ...
}
```



Caller's AR

Activation Record Layout for C



```
...
f(1,2,3) ;
...
void f(int a, int b, int c) {
    int x, y, z ;
    ...
}
```

Caller's AR

preserved %eax

preserved %edx

preserved %ecx

Caller-saved registers

c : 3 16(%ebp)

b : 2 12(%ebp)

a : 1 8(%ebp)

return addr

current %ebp → previous %ebp

local var x -4(%ebp)

local var y -8(%ebp)

local var z -12(%ebp)

preserved %ebx

preserved %esi

preserved %edi

Callee-saved registers

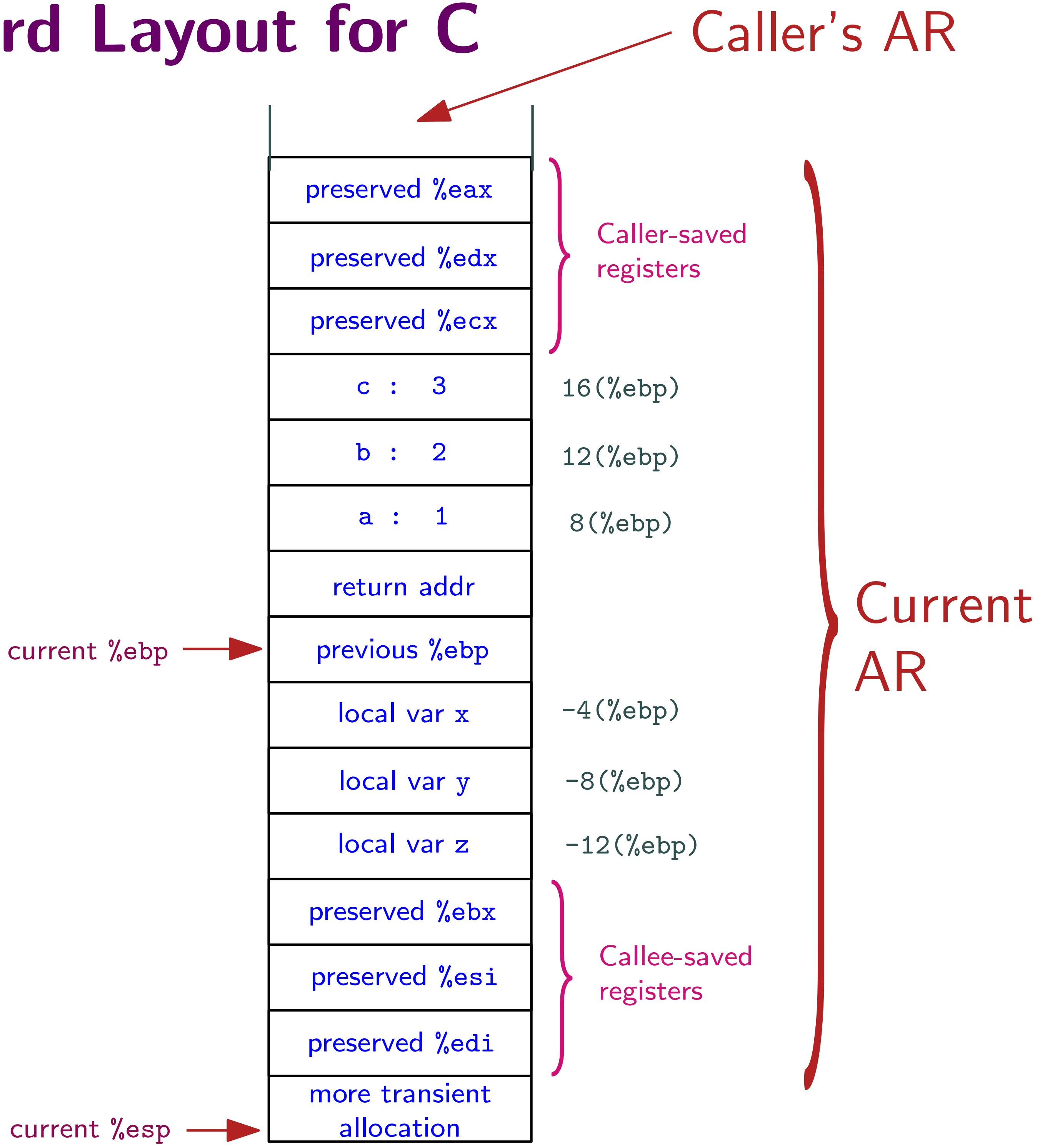
current %esp → more transient allocation

Current AR

Activation Record Layout for C

```
...
f(1,2,3) ;
...
void f(int a, int b, int c) {
    int x, y, z ;
    ...
}
```

```
    pushl %eax
    pushl %edx
    pushl %ecx
    pushl $3
    pushl $2
    pushl $1
    call f
    addl $8,%esp
    popl %ecx
    popl %edx
    popl %eax
    ........


f:
    pushl %ebp
    movl %esp,%ebp
    subl $12,%esp
    pushl %ebx
    pushl %esi
    pushl %edi

    ........

    popl %edi
    popl %esi
    popl %ebx
    movl %ebp,%esp
    popl %ebp
    ret

    .........
```



Caller's AR

preserved %eax

preserved %edx

preserved %ecx

Caller-saved registers

c : 3 — 16(%ebp)

b : 2 — 12(%ebp)

a : 1 — 8(%ebp)

return addr

current %ebp → previous %ebp

local var x — -4(%ebp)

local var y — -8(%ebp)

local var z — -12(%ebp)

preserved %ebx

preserved %esi

preserved %edi

Callee-saved registers

current %esp → more transient allocation

Current AR

Recursive Factorial

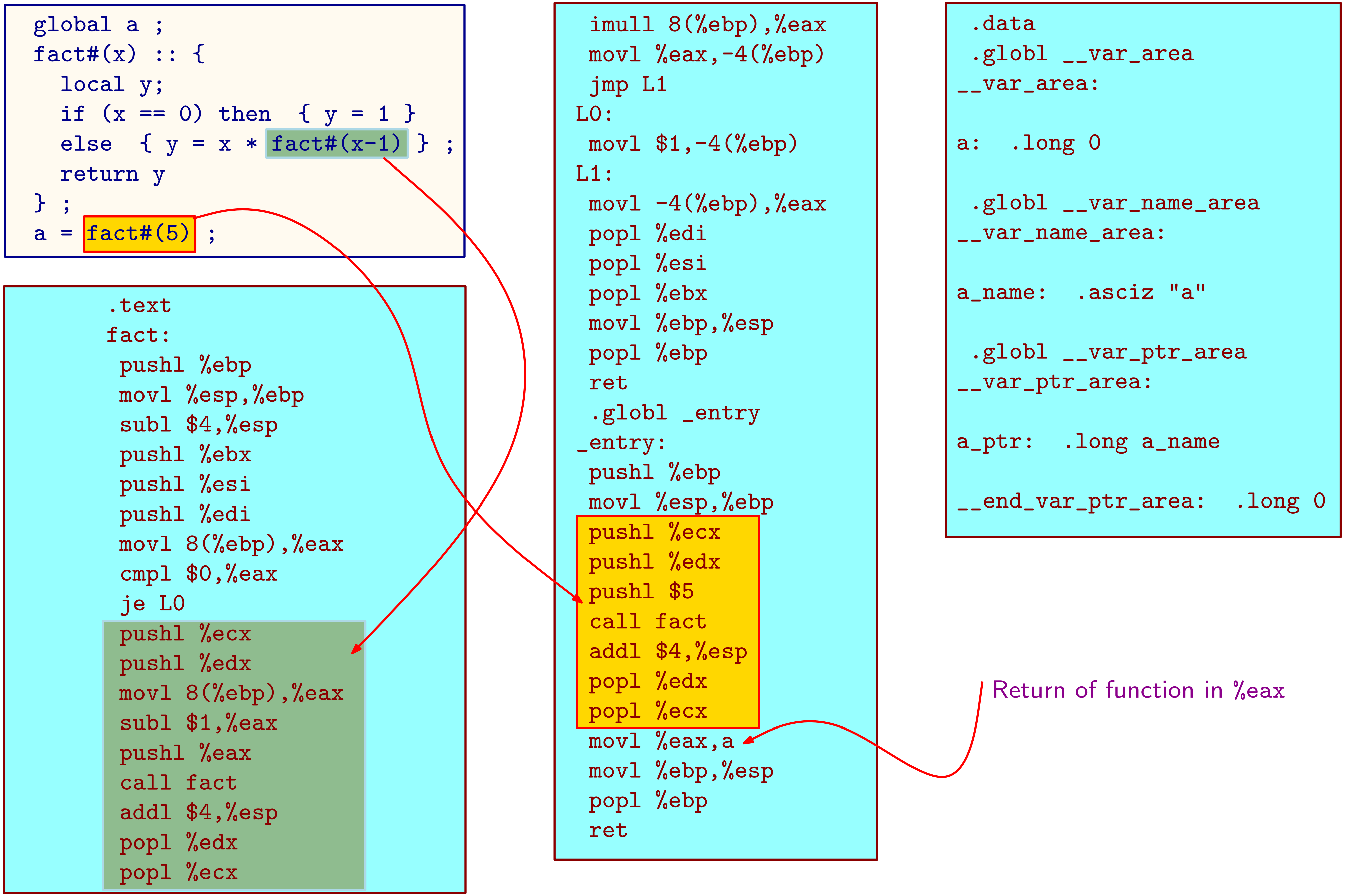
```
global a ;
fact#(x) :: {
  local y;
  if (x == 0) then  { y = 1 }
  else  { y = x * fact#(x-1) } ;
  return y
} ;
a = fact#(5) ;
```

```
.text
fact:
 pushl %ebp
 movl %esp,%ebp
 subl $4,%esp
 pushl %ebx
 pushl %esi
 pushl %edi
 movl 8(%ebp),%eax
 cmpl $0,%eax
 je L0
 pushl %ecx
 pushl %edx
 movl 8(%ebp),%eax
 subl $1,%eax
 pushl %eax
 call fact
 addl $4,%esp
 popl %edx
 popl %ecx
```

```
 imull 8(%ebp),%eax
 movl %eax,-4(%ebp)
 jmp L1
L0:
 movl $1,-4(%ebp)
L1:
 movl -4(%ebp),%eax
 popl %edi
 popl %esi
 popl %ebx
 movl %ebp,%esp
 popl %ebp
 ret
 .globl _entry
_entry:
 pushl %ebp
 movl %esp,%ebp
 pushl %ecx
 pushl %edx
 pushl $5
 call fact
 addl $4,%esp
 popl %edx
 popl %ecx
 movl %eax,a
 movl %ebp,%esp
 popl %ebp
 ret
```

```
 .data
 .globl __var_area
__var_area:

a:   .long 0

 .globl __var_name_area
__var_name_area:

a_name:   .asciz "a"

 .globl __var_ptr_area
__var_ptr_area:

a_ptr:   .long a_name

__end_var_ptr_area:   .long 0
```

Recursive Factorial



```
global a ;
fact#(x) :: {
  local y;
  if (x == 0) then  { y = 1 }
  else  { y = x * fact#(x-1) } ;
  return y
} ;
a = fact#(5) ;
```

```
      .text
      fact:
       pushl %ebp
       movl %esp,%ebp
       subl $4,%esp
       pushl %ebx
       pushl %esi
       pushl %edi
       movl 8(%ebp),%eax
       cmpl $0,%eax
       je L0
       pushl %ecx
       pushl %edx
       movl 8(%ebp),%eax
       subl $1,%eax
       pushl %eax
       call fact
       addl $4,%esp
       popl %edx
       popl %ecx
```

```
 imull 8(%ebp),%eax
 movl %eax,-4(%ebp)
 jmp L1
L0:
 movl $1,-4(%ebp)
L1:
 movl -4(%ebp),%eax
 popl %edi
 popl %esi
 popl %ebx
 movl %ebp,%esp
 popl %ebp
 ret
 .globl _entry
_entry:
 pushl %ebp
 movl %esp,%ebp
 pushl %ecx
 pushl %edx
 pushl $5
 call fact
 addl $4,%esp
 popl %edx
 popl %ecx
 movl %eax,a
 movl %ebp,%esp
 popl %ebp
 ret
```

```
 .data
 .globl __var_area
__var_area:

a:   .long 0

 .globl __var_name_area
__var_name_area:

a_name:   .asciz "a"

 .globl __var_ptr_area
__var_ptr_area:

a_ptr:   .long a_name

__end_var_ptr_area:   .long 0
```

Return of function in %eax

Compilation of Procedure Definition

```
cs((P#L::{S}),Code,Ain,Aout) :- !, % Generate code for procedure definition
            % Generate procedure label
    ProcLabel = ['\n',P,':'],
            % Preserve the original attributes, and restore at end of block
    get_assoc(local_vars,Ain,OriginalLocalVars),
    get_assoc(top_local_vars,Ain,OriginalTopLocalVars,Atlv,0),
    get_assoc(max_local_vars,Atlv,OriginalMaxLocalVars,Amlv,0),
    put_assoc(top_args,Amlv,4,Ata),
            % Process formal arguments
    proc_args(L,Ata,Apa),
            % compile the procedure body (may contain local variable declarations)
    cs({S},CodeS,Apa,Acs),
            % retrieve the amount of space used for local variables
    get_assoc(max_local_vars,Acs,Max),
            % Generate procedure's prologue, which saves the old frame pointer
            % and loads the frame pointer register with the current top of the stack.
            % After execution of this code, all arguments can be referred to via their
            % mappings stored in the local variables attribute
    Prologue = [ '\n\t\t pushl %ebp',
                 '\n\t\t movl %esp,%ebp' ],
            % Check if allocation needed for local variables and generate adequate code
    (    Max == 0 -> AllocCode = [] ; AllocCode = [ '\n\t\t subl $',Max,',%esp' ] ),
```

Compilation of Procedure Definition

```
        % Registers %ebx, %esi, %edi are callee saved. The procedure should preserve
        % their original values. We save them unconditionally, which is not very efficient.
        % A better alternative would be to check the code for the procedure's body, and
        % save them only if they are used there.
CalleeSaved = [ '\n\t\t pushl %ebx\n\t\t pushl %esi\n\t\t pushl %edi' ],
        % What is saved needs to be restored
CalleeRestored = [ '\n\t\t popl %edi\n\t\t popl %esi\n\t\t popl %ebx' ],
        % The epilogue restores the frame pointer to its original value, and returns
        % to the caller
Epilogue = [ '\n\t\t movl %ebp,%esp',
             '\n\t\t popl %ebp',
             '\n\t\t ret'],
% lay out the code
append([ProcLabel,Prologue,AllocCode,CalleeSaved,CodeS,CalleeRestored,Epilogue],Code),
% Restore saved attributes
put_assoc(top_args,Acs,none,Atopa),
put_assoc(local_vars,Atopa,OriginalLocalVars,Alv),
put_assoc(max_local_vars,Alv,OriginalMaxLocalVars,Aomlv),
put_assoc(top_local_vars,Aomlv,OriginalTopLocalVars,Aout).
```

Support Predicates: proc_args

```
% Helper that would map each formal argument of a procedure
% into an offset N, so that the variable can be referred as
% N(%ebp) later in the code. Called by 'proc_args'.
proc_args_helper(V,Ain,Aout) :-
    get_assoc(top_args,Ain,Tin,A0,Tout),
    get_assoc(local_vars,A0,VS,Aout,[(V,Ref)|VS]),
    Tout #= Tin + 4, atomic_list_concat([Tout,'(%ebp)'],Ref).

% Predicate that iterates through a list of identifiers,
% assumed to be the list of formal arguments of a procedure,
% calling 'proc_args_helper' on each of them. The end result
% is that all the arguments will appear in the local symbol
% table, with the corresponding mappings, ready to be referenced
% throughout the compilation of the current scope.
proc_args((VH,VT),Ain,Aout) :- !,
    proc_args_helper(VH,Ain,Aaux),
    proc_args(VT,Aaux,Aout).
proc_args(V,Ain,Aout) :- !,
    V =.. L, L \= [_,_|_],
    proc_args_helper(V,Ain,Aout).
```

Return Statement

```
cs(return X, Code, Ain, Aout) :- !, % evaluates expression X and loads result into %eax
    put_assoc(context,Ain,expr,A1),
    ce(X,CX,A1,Aout), % Don't call 'comp_expr', try to enforce result into %eax through constraint solving
    get_assoc(expr_result,Aout,Result),
    % Figure out where the result is, and generate correct transfer instruction
    % If result in register, unify first element with 0, in attempt to end up with
    % result right there, and save on the transfer instruction
    (  Result = [0|_], Result ins 0..5, label(Result), Res = [],! ;
       (Result = const(T),atomic_concat('$',T,N) ; Result = [R|_],N=reg32(R) ; Result = id(N)),!,
        Res = ['\n\t\t movl ',N,',%eax'] ),
    % registers are still in reg32(X) form, need to be translated
    replace_regs(CX,C),
    % lay out the code
    append(C,Res,Code).
```

- Simplification: expected to appear only at the end of the procedure.
- May take any expression as argument
- May be missing is the procedure returns no value (i.e. is used as a statement)

Compilation of Procedure Call

```
ce(P#L,Code,Ain,Aout) :- % generate code for procedure call, which is in fact an _expression_
                % Arguments must be pushed on the stack in reverse order,
                % so we reverse the list of actual args, if it's
                % not empty or singleton
    (    L = (H,T) -> rev(T,H,LR) ; LR = L ),
                % Code to save caller saved registers
    CallerSaved = [ '\n\t\t pushl %ecx',
                    '\n\t\t pushl %edx' ],
                % Generate code to evaluate each argument in LR (arguments
                % are expressions) and push it on the stack
    push_args(LR,ArgC,Ain,A0,R), RR #= R*4 ,
                % Arguments will have to be cleared by caller upon return,
                % To that end, the numnber of args is computed in R,
                % and RR is the number of bytes to clear from the stack
                % in order to deallocate the storage for the arguments

                % The instruction that calls the procedure
    Call = [ '\n\t\t call ',P ],

                % The instruction to clear arguments from the stack
    ResC = [ '\n\t\t addl $',RR,',%esp' ],

                % Code to restore the caller-saved registers
    CallerRestored = [ '\n\t\t popl %edx',
                       '\n\t\t popl %ecx' ],
```

Compilation of Procedure Call

- Procedure calls are in fact expressions, so they must be implemented in the expression compiler
- We have to be compatible with the rest of the expression compiler by computing a residue, and indicating that the result is in %eax

```
get_assoc(context,Ain,Ctx),
            % Generate the adequate residue
(    Ctx = expr -> Residue = [] ; Residue = [ '\n\t\t cmpl $0,%eax' ] ),
            % Lay out the code
append([CallerSaved,ArgC,Call,ResC,CallerRestored],Code),
            % Record in the attributes that the result is in %eax, and
            % since all other registers are restored to original values
            % then we don't need to specify any other registers as "used"
put_assoc(expr_result,A0,[0],Aout).
```

Support Predicates: push_args and rev

```
% Reverse a list made up from pairs of pairs. Useful to reverse
% the list of arguments of a procedure, when the arguments are
% about to be pushed on the stack. Assume that the list of args
% has the form (First,Rest). Then the call should be:
%
%           rev(Rest,First,Reversed)
%
rev((X,Y),L,R) :- !, rev(Y,(X,L),R).
rev(X,L,(X,L)).

% Procedure to generate code that pushes a list of
% arguments on the stack. The list of arguments is
% assumed to be already reversed.
push_args((X,Y),Code,Ain,Aout,Lgth) :- !,
    put_assoc(context,Ain,expr,A0),
    comp_expr(X,CX,A0,A1),
    push_result(A1,PushX),
    push_args(Y,CY,A1,Aout,LY),
    append([CX,PushX,CY],Code),
    Lgth #= LY + 1.
push_args(void,[],A,A,0) :- !.
push_args(X,Code,Ain,Aout,1):-
    put_assoc(context,Ain,expr,A0),
    comp_expr(X,CX,A0,Aout),
    push_result(Aout,PushC),
    append(CX,PushC,Code).
```

Procedure Call as Statement

```
cs((P#L),Code,Ain,Aout) :- !,       % Procedure call as statement (and not as expression)
    put_assoc(context,Ain,expr,A0), % Compile as expression, do not store value.
    comp_expr((P#L),Code,A0,Aout).
```

Test Thoroughly

```
global a,b,c,d,e,ff,gg ;

facttc#(x,y) :: {
  local z;
  if (x == 0) then  { z = y }
              else  { z = facttc#(x-1,y*x) } ;
  return z
} ;

fibtc#(n,x,y) :: {
  local z ;
  if n == 0 then { z = y }
             else { z = fibtc#(n-1,x+y,x) }
} ;

f#(a,b,c) :: {
  local x,y,z ;
  x = a + b ;
  y = a + c ;
  z = b + c ;
  if ( (0 < x) /\ (x < 100) /\ (0 < y)
       /\ (y < 100) /\ (0 < z) /\ (z < 100) )
     then { x = f#(x,y,z) + f#(a-b,a-c,b-c) }
     else { x = 1 } ;
  return x
} ;
```

```
g#(void) :: {
    return a + 1
};

b = 5 ; c = 3 ;

a = facttc#(b,1) * facttc#(c,1) ;

c = facttc#(b+c,1);

b = fibtc#(facttc#(3,1),1,0)/fibtc#(3,1,0);

d = fibtc#( facttc#(2,1)*facttc#(2,1)+b,
            facttc#(10,2)/facttc#(10,1),0) ;

e = fibtc#(6,1,0) ;

ff = f#(10,20,30)*f#(1,1,1) ;

gg = g#(void)
```

Generated code too large to show!

Pascal Calling Convention

- Actual arguments are pushed on the stack in left-to-right order

- Caller responsible for cleaning up arguments from the stack

- Processor has instruction `ret N` to assist with this
  - Returns, and then increases `%esp` by `N`

- Not possible to have variable number of arguments for procedure

- Available for C procedures via the `pascal` modifier

- Many other conventions available; check out ”x86 calling conventions” on Wikipedia

Conclusion

- Procedures require activation records stacked up to allow recursion
- Multiple calling conventions; the C calling convention allows variable number of arguments (which we have not implemented)
- In general, with nested procedures and procedural abstraction, the AR can become more complicated