Topic 6: Activation Records

COS 320

Compiling Techniques

Princeton University
Spring 2016

Lennart Beringer

Activation Records

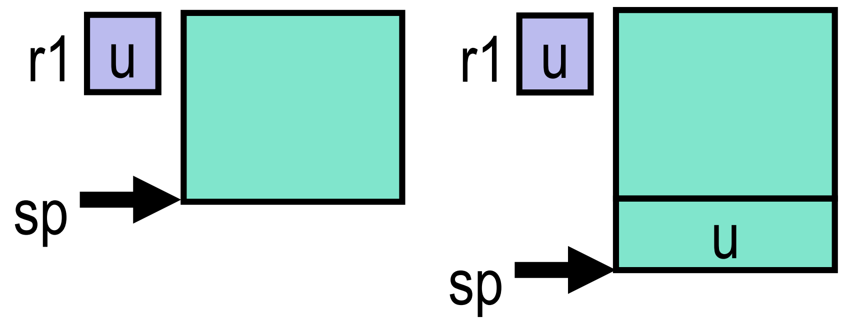
- Modern imperative programming languages typically have *local* variables.
  - Created upon entry to function.
  - Destroyed when function returns.
- Each invocation of a function has its own *instantiation* of local variables.
  - Recursive calls to a function require several instantiations to exist simultaneously.
  - Functions return only after all functions it calls have returned ⇒ last-in-first-out (LIFO) behavior.
  - A LIFO structure called a *stack* is used to hold each instantiation.
- The portion of the stack used for an invocation of a function is called the function's *stack frame* or *activation record*.

The Stack

- holds local variables (and other data, see later)
- implemented as large array that typically grows downwards towards lower addresses, and shrinks upwards

Push(r1):

```
stack_pointer --;
M[stack_pointer] = r1;
```



r1 = Pop():

```
r1 = M[stack_pointer];
stack_pointer++;
```


But: occasionally, we also need to access the previous activation record (ie frame of caller). Hence, simple push/pop insufficient.

Solution:
- treat stack as array with index off of stack_pointer
- push/pop entire activation records

The Stack

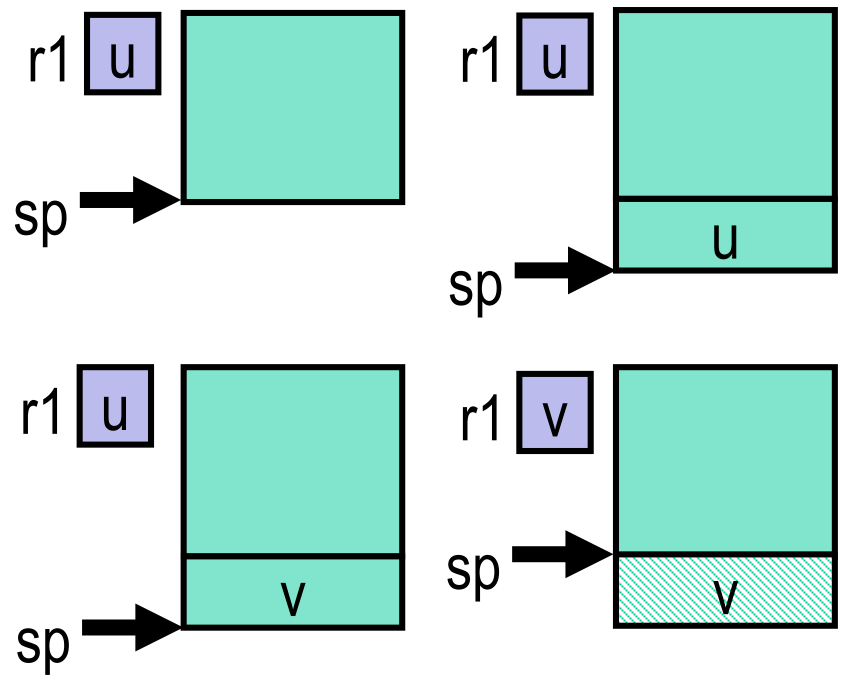
- holds local variables (and other data, see later)
- implemented as large array that typically grows downwards towards lower addresses, and shrinks upwards

Push(r1):

```
stack_pointer --;
M[stack_pointer] = r1;
```

r1 = Pop():

```
r1 = M[stack_pointer];
stack_pointer++;
```



But: occasionally, we also need to access the previous activation record (ie frame of caller). Hence, simple push/pop insufficient.

Solution:
- treat stack as array with index off of stack_pointer
- push/pop entire activation records

The Stack

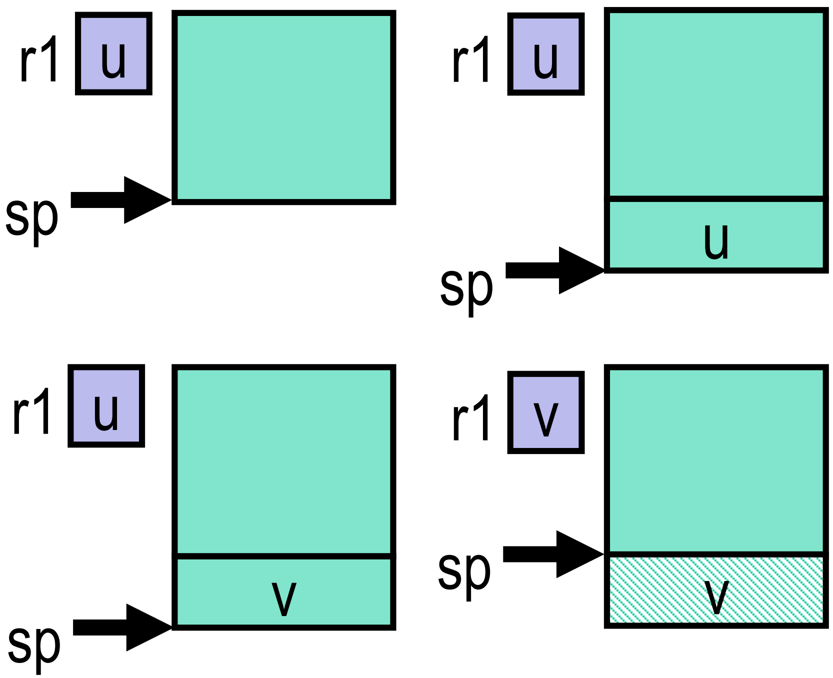
- holds local variables (and other data, see later)
- implemented as large array that typically grows downwards towards lower addresses, and shrinks upwards

Push(r1):

```
stack_pointer --;
M[stack_pointer] = r1;
```

r1 = Pop():

```
r1 = M[stack_pointer];
stack_pointer++;
```



But: occasionally, we also need to access the previous activation record (ie frame of caller). Hence, simple push/pop insufficient.

Solution:

- treat stack as array with index off of stack_pointer
- push/pop entire activation records

Example

Consider:

```
let
  function g(x:int) =
    let
      var y := 10
    in
      x + y
    end
  function h(y:int):int =
    y + g(y)
in
  h(4)
end
```

Example

Step 1: `h(4)` **called**

Chunk of memory allocated on the stack in order to hold local variables of h. The activation record (or stack frame) of h is pushed onto the stack.

Stack Frame for h	y=4

Step 2: `g(4)` **called**

Activation record for g allocated (pushed) on stack.

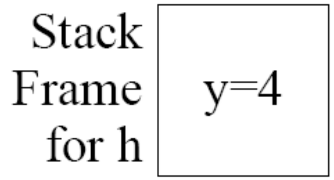
Stack Frame for h	y=4
Stack Frame for g	x=4 y=10

```
let
  function g(x:int) =
    let
      var y := 10
    in
      x + y
    end
  function h(y:int):int =
    y + g(y)
in
  h(4)
end
```

Example

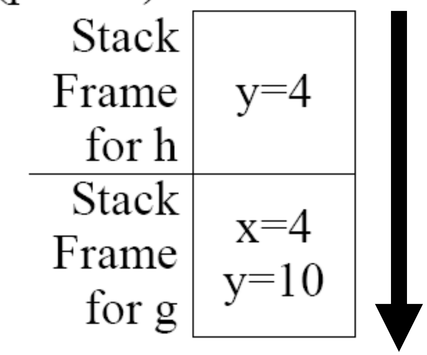
Step 1: `h(4)` **called**

Chunk of memory allocated on the stack in order to hold local variables of h. The activation record (or stack frame) of h is pushed onto the stack.



Step 2: `g(4)` **called**

Activation record for g allocated (pushed) on stack.




```
let
  function g(x:int) =
    let
      var y := 10
    in
      x + y
    end
  function h(y:int):int =
    y + g(y)
in
  h(4)
end
```

Example

Step 3: `g(4)` **returns with value 14**

Activation record for g deallocated (popped) from stack.

Stack Frame for h	
	y=4
	rv = 14

Step 4: `h(4)` **returns with value 18**

Activation record for h deallocated (popped) from stack. Stack now empty.

```
let
  function g(x:int) =
    let
      var y := 10
    in
      x + y
    end
  function h(y:int):int =
    y + g(y)
in
  h(4)
end
```

Example

Step 3: `g(4)` **returns with value 14**

Activation record for g deallocated (popped) from stack.

Stack Frame for h	y=4 rv = 14

Step 4: `h(4)` **returns with value 18**

Activation record for h deallocated (popped) from stack. Stack now empty.

```
let
  function g(x:int) =
    let
      var y := 10
    in
      x + y
    end
  function h(y:int):int =
    y + g(y)
in
  h(4)
end
```

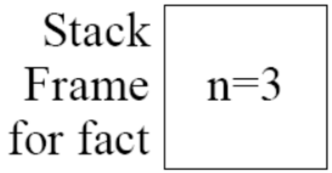
Recursive Example

Can have multiple stack frames for same function (different invocations) on stack at any given time due to recursion.

Consider:

```
let
  function fact(n:int):int =
    if n = 0 then 1
    else n * fact(n - 1)
in
  fact(3)
end
```

Step 1: Record for `fact(3)` **pushed on stack.**



Recursive Example

Step 2: Record for `fact(2)` pushed on stack.

Stack Frame for fact	n=3
Stack Frame for fact	n=2

Step 3: Record for `fact(1)` pushed on stack.

Stack Frame for fact	n=3
Stack Frame for fact	n=2
Stack Frame for fact	n=1

```
let
  function fact(n:int):int =
    if n = 0 then 1
    else n * fact(n - 1)
in
  fact(3)
end
```

Recursive Example

Step 2: Record for `fact(2)` pushed on stack.

Stack Frame for fact	n=3
Stack Frame for fact	n=2

Step 3: Record for `fact(1)` pushed on stack.

Stack Frame for fact	n=3
Stack Frame for fact	n=2
Stack Frame for fact	n=1

```
let
  function fact(n:int):int =
    if n = 0 then 1
    else n * fact(n - 1)
in
  fact(3)
end
```

Recursive Example

```
let
   function fact(n:int):int =
      if n = 0 then 1
      else n * fact(n - 1)
in
   fact(3)
end
```

Step 4: Record for `fact(0)` pushed on stack.

Stack Frame for fact	n=3
Stack Frame for fact	n=2
Stack Frame for fact	n=1
Stack Frame for fact	n=0

Step 5: Record for `fact(0)` popped off stack, 1 returned.
Step 6: Record for `fact(1)` popped off stack, 1 returned.
Step 7: Record for `fact(2)` popped off stack, 2 returned.
Step 8: Record for `fact(3)` popped off stack, 3 returned. Stack now empty.

Functional Languages

In some functional languages (such as ML, Scheme), local variables cannot be stored on stack.

```
fun f(x) =
  let
    fun g(y) = x + y
  in
    g
  end
```

Consider:

```
- val z = f(4)
- val w = z(5)
```

Assume variables are stack-allocated.

Functional Languages

Step2

Step 1: `f(4)` called.

Frame for `f(4)` pushed, `g` returned, frame for `f(4)` popped.

Stack Frame for f	x=4

Stack empty.

Step 3: `z(5)` called

Stack Frame for z	y=5

Memory location containing x has been deallocated!

```
fun f(x) =
  let
    fun g(y) = x + y
  in
    g
  end
```

Consider:

```
- val z = f(4)
- val w = z(5)
```

Functional Languages

Step2

Step 1: `f(4)` **called.**

Frame for `f(4)` pushed, `g` returned, frame for `f(4)` popped.

Stack Frame for f: x=4

Stack empty.

Step 3: `z(5)` **called**

i.e. g(5)

Stack Frame for z: y=5

Memory location containing x has been deallocated!

```
fun f(x) =
  let
    fun g(y) = x + y
  in
    g
  end
```

Consider:

```
- val z = f(4)
- val w = z(5)
```

Functional Languages

Combination of

- nested functions and
- functions that are returned as results (i.e. higher-order)

requires that

- local variable remain in existence even after enclosing function has returned
- activation records are allocated on heap (“closures”), not on the stack

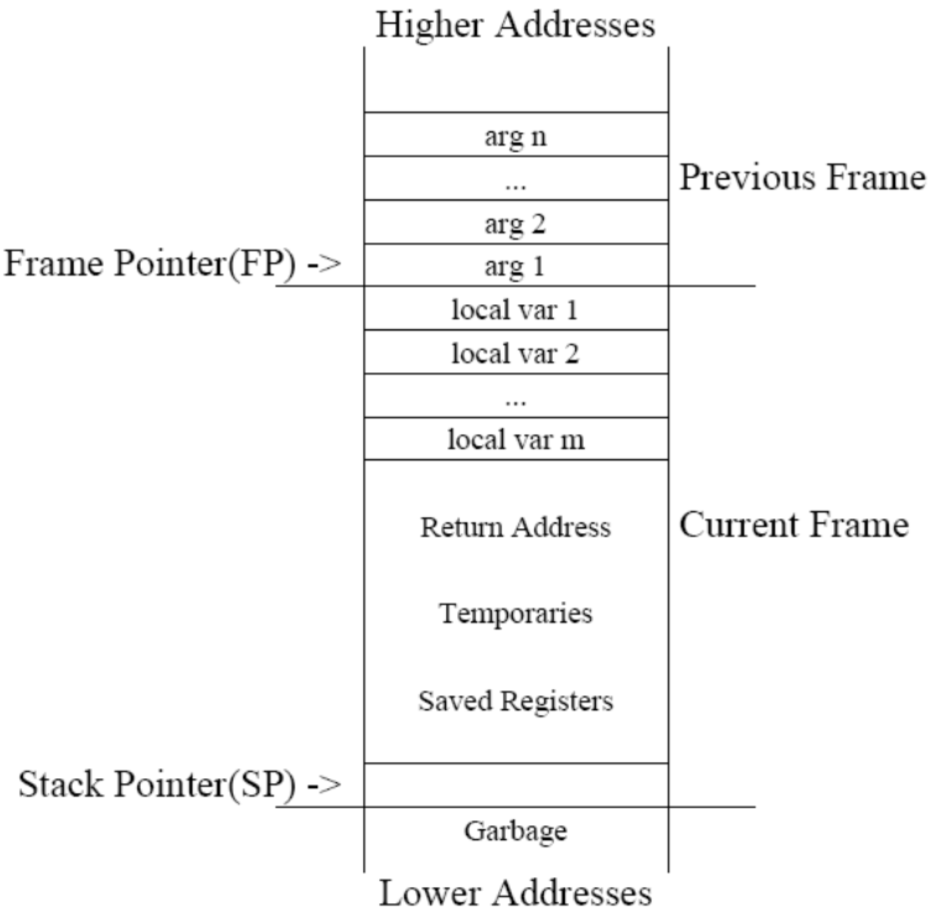
For now, focus on languages that use stack.

Stack Frame Organizations

How is data organized in stack frame?

- Compiler can use any layout scheme that is convenient.
- Microprocessor manufactures specify "standard" layout schemes used by all compilers.
  - Sometimes referred to as *Calling Conventions*.
  - If all compilers use the same calling conventions, then functions compiled with one compiler can call functions compiled with another.
  - Essential for interaction with OS/libraries.

Typical Stack Frame



Callee can access arguments by offset from FP:

```
argument 1: M[FP]
argument 2: M[FP + 1]
```

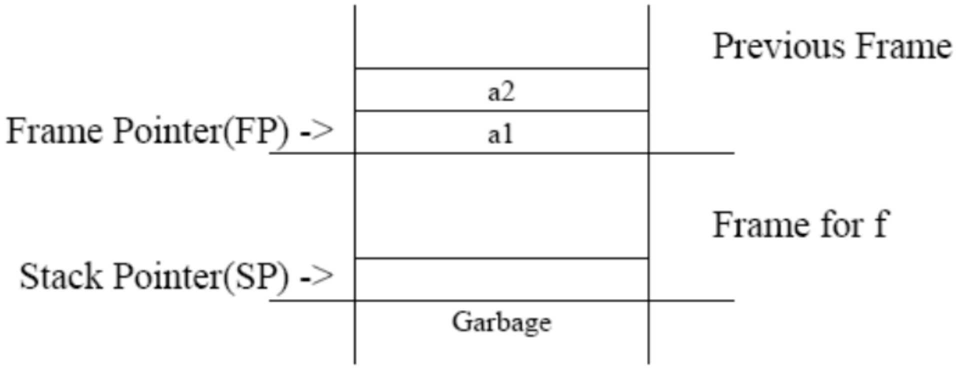
Local variables accessed by offset from FP:

```
local variable 1: M[FP - 1]
local variable 2: M[FP - 2]
```

Stack Frame Example

Suppose `f(a1, a2)` calls `g(b1, b2, b3)`

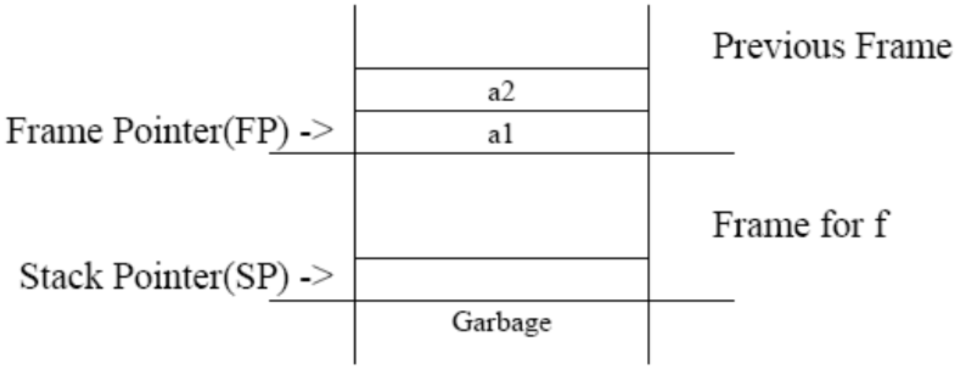
Step 1:



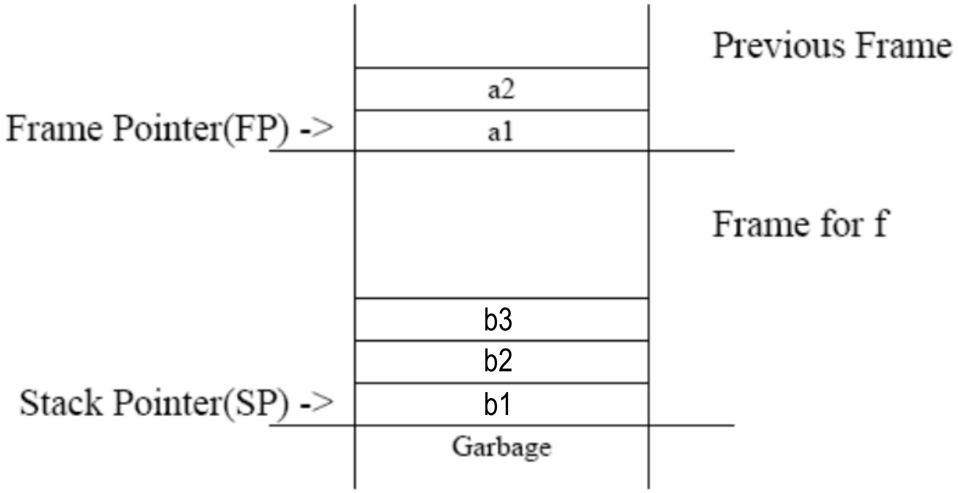
Stack Frame Example

Suppose `f(a1, a2)` calls `g(b1, b2, b3)`

Step 1:



Step 2: push outgoing arguments; decrease SP

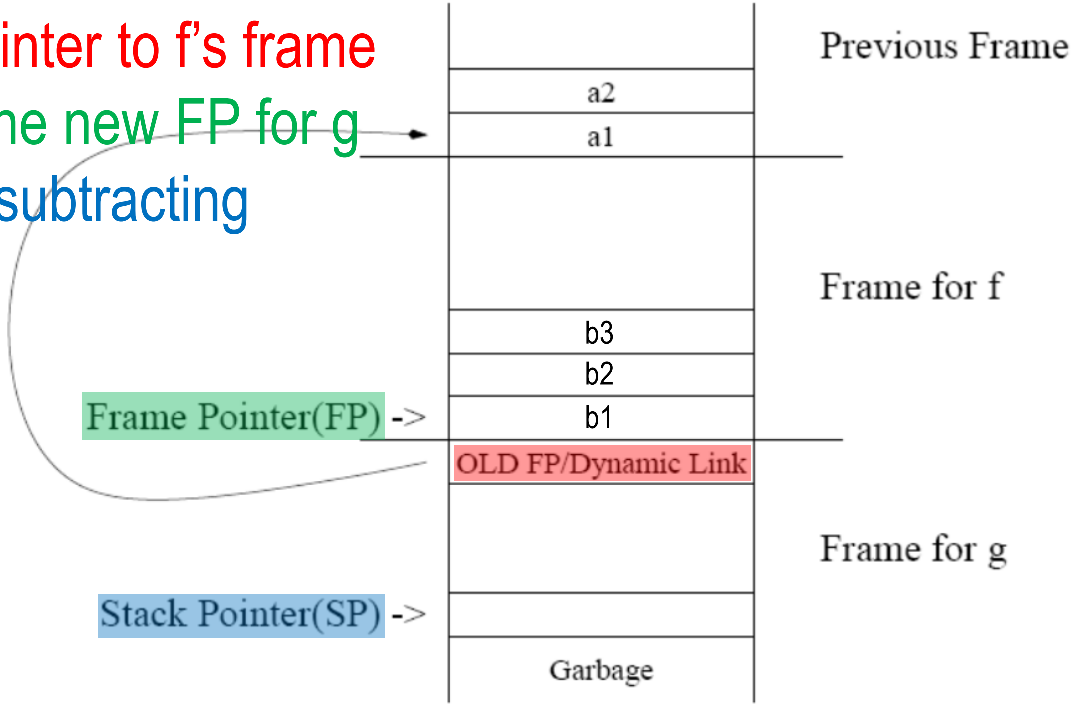


Stack Frame Example

Suppose `f(a1, a2)` calls `g(b1, b2, b3)`

Step 3:

- push frame pointer to f's frame
- make old SP the new FP for g
- update SP by subtracting size(g)



Dynamic link (AKA Control link) points to the activation record of the caller.

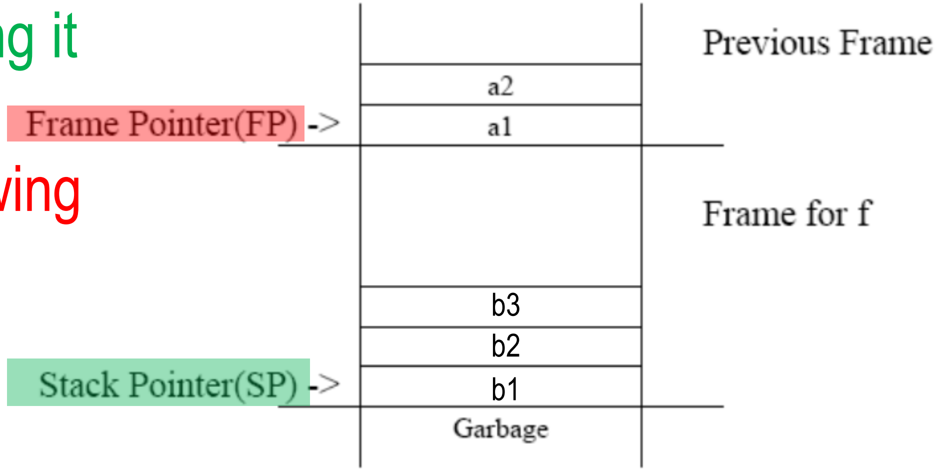
- Optional if size of caller activation record is known at compile time.
- Used to restore stack pointer during return sequence.

Stack Frame Example

Suppose `f(a1, a2)` calls `g(b1, b2, b3)`, and returns.

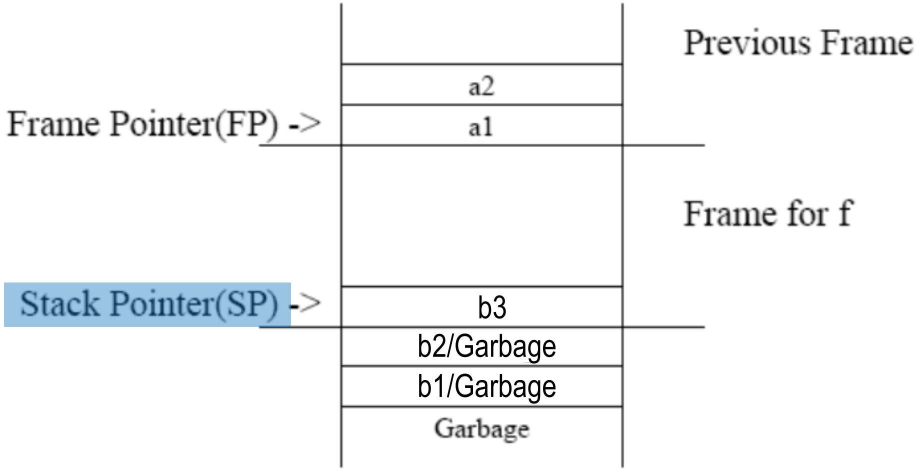
Step 4

- restore f's SP by setting it to g's FP
- restore f's FP by following g's dynamic link, now located at SP-1



Step 5

- pop the arguments by incrementing SP



Parameter Passing

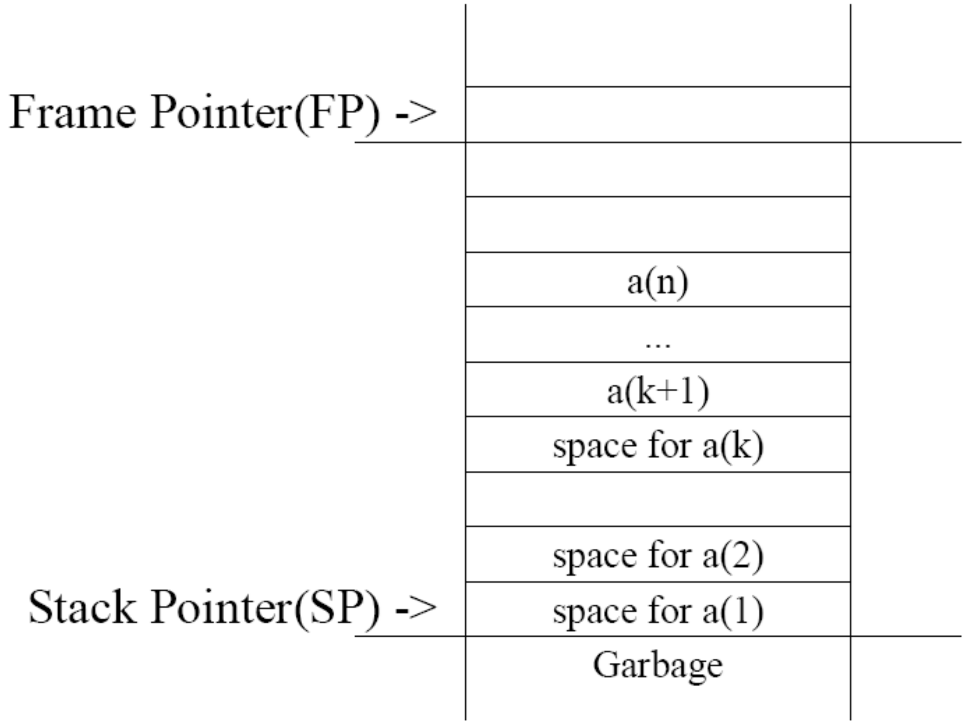
`f(a`$_1$`, a`$_2$`, ..., a`$_n$`)`

- Registers are faster than memory.
- Compiler should keep values in register whenever possible.
- Modern calling convention: rather than placing a_1, a_2, ..., a_n on stack frame, put a_1, ..., a_k ($k = 4$) in registers r_p, r_{p+1}, r_{p+2}, r_{p+3} and a_{k+1}, a_{k+2}, a_{k+2}, ..., a_n in memory
- If r_p, r_{p+1}, r_{p+2}, r_{p+3} are needed for other purposes, callee function must save incoming argument(s) in stack frame.
- C language allows programmer to take address of formal parameter and guarantees that formals are located at consecutive memory addresses.
  - If a register argument has its address taken, it must be written into stack frame.
  - Saving it in "saved registers" area of stack won't make it consecutive with memory resident arguments.
  - Space must be allocated even if parameters are passed through register.

  Solution: space is reserved by caller, but only written to by callee, and only if necessary

Parameter Passing

If register argument has address taken, *callee* writes register into corresponding space.



Registers

Registers hold:

- Some Parameters
- Return Value
- Local Variables
- Intermediate results of expressions (temporaries)

Stack Frame holds:

- Variables passed by reference or have their address taken (&)
- Variables that are accessed by procedures nested within current one.
- Variables that are too large to fit into register file.
- Array variables (address arithmetic needed to access array elements).
- Variables whose registers are needed for a specific purpose (parameter passing)
- *Spilled* registers. Too many local variables to fit into register file, so some must be stored in stack frame.

Registers

Compilers typically place variables on stack until it can determine whether or not it can be promoted to a register (e.g. no references).

The assignment of variables to registers is done by the *Register Allocator*.

Registers

Register state for a function must be saved before a callee function can use them.

Calling convention describes two types of registers.

- *Caller-save* register are the responsibility of the calling function.
  - Caller-save register values are saved to the stack by the calling function if they will be used after the call.
  - The callee function can use caller-save registers without saving their original values.
- *Callee-save* registers are the responsibility of the called function.
  - Callee-save register values must be saved to the stack by called function before they can be used.
  - The caller (calling function) can assume that these registers will contain the same value before and after the call.

Placement of values into callee-save vs. caller-save registers is determined by the register allocator.

Return Address and Return Value

A called function must be able to return to calling function when finished.

- Return address is address of instruction following the function call.
- Return address can be placed on stack or in a register.
- The *call* instruction in modern machines places the return address in a designated register.
- This return address is written to stack by callee function in non-leaf functions.

Return value is placed in designated register by callee function.

Frame Resident Variables

- A variable *escapes* and must hence be held in memory if
  - it is passed by reference,
  - its address is taken, or
  - it is accessed from a nested function.
- Variables cannot be assigned a location at declaration time.
  - Escape conditions not known.
  - Assign provisional locations, decide later if variables can be promoted to registers.
- `escape` set to true by default.

Static Links

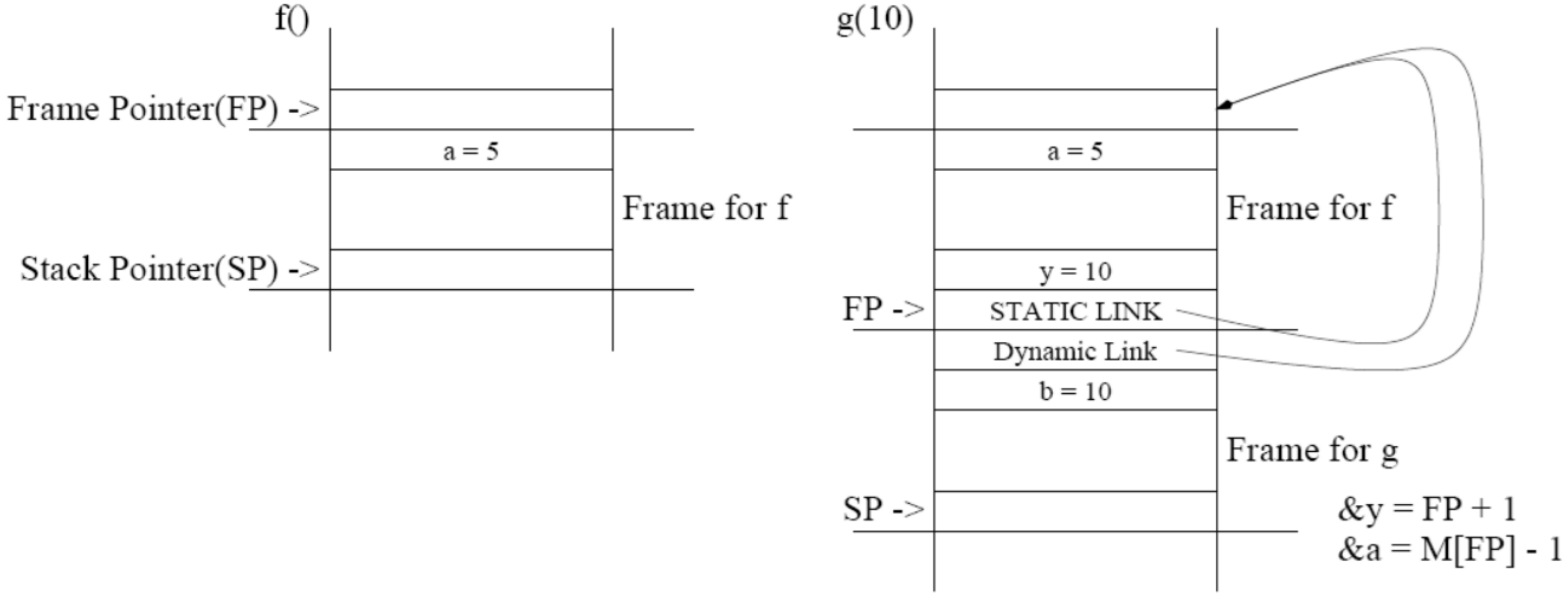
In languages that allow nested functions (e.g. Tiger), functions must access outer function's stack frame.

```
let
  function f():int = let
      var a:=5
      function g(y:int):int = let
          var b:=10
          function h(z:int):int =
            if z > 10 then h(z / 2)
            else z + b * a                 <- b, a of outer fn
        in
          y + a + h(16)                    <- a of outer fn
        end
    in
      g(10)
    end
in f() end
```

Static Links

Solution:

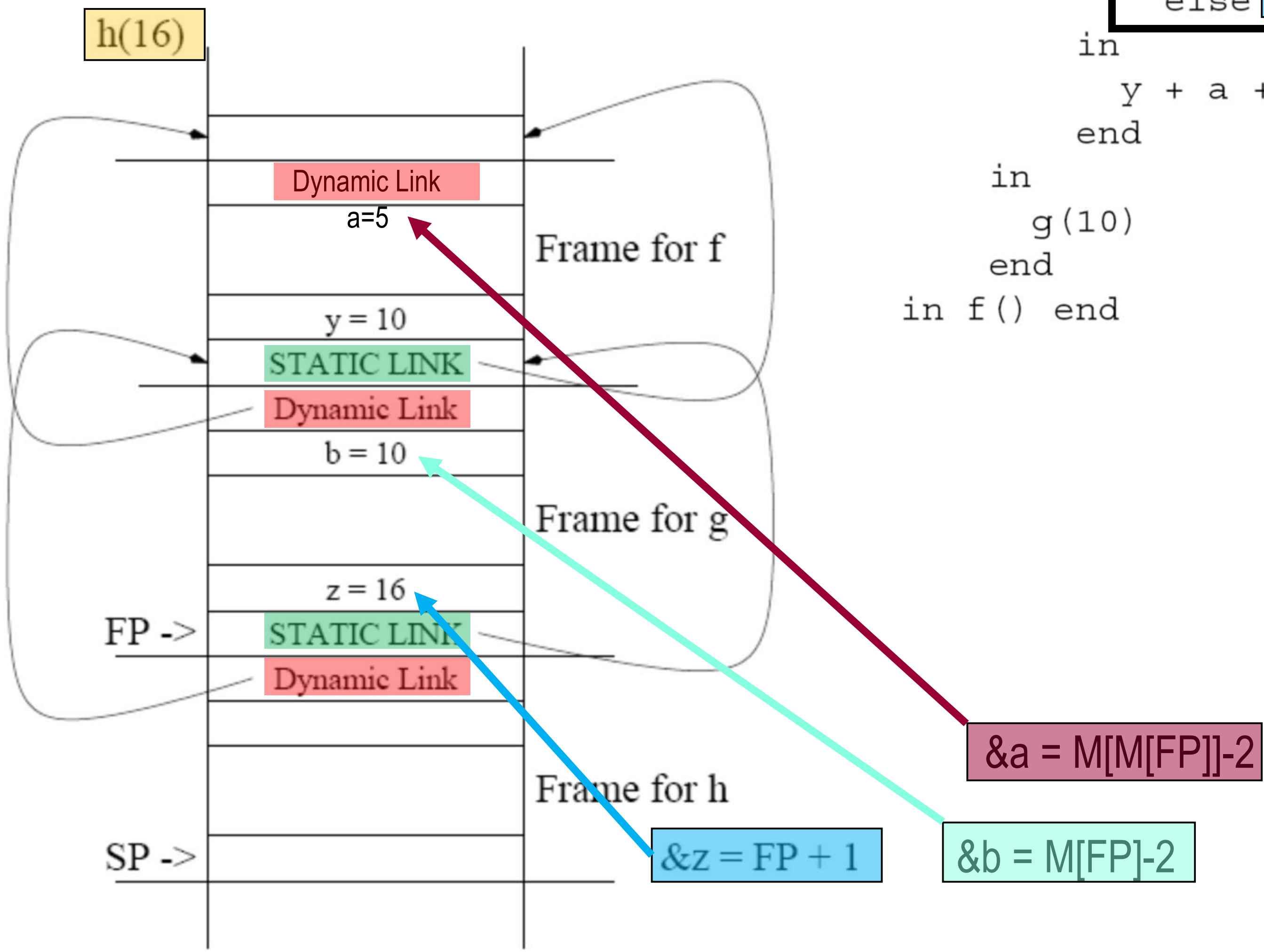
Whenever f is called, it is passed pointer to most recent activation record of g that immediately encloses f in program text ⇒ Static Link (AKA Access Link).



Static Links: nonrecursive call

- Dynamic links point to FP of caller
- Static links point to FP of surrounding function's most recent instance

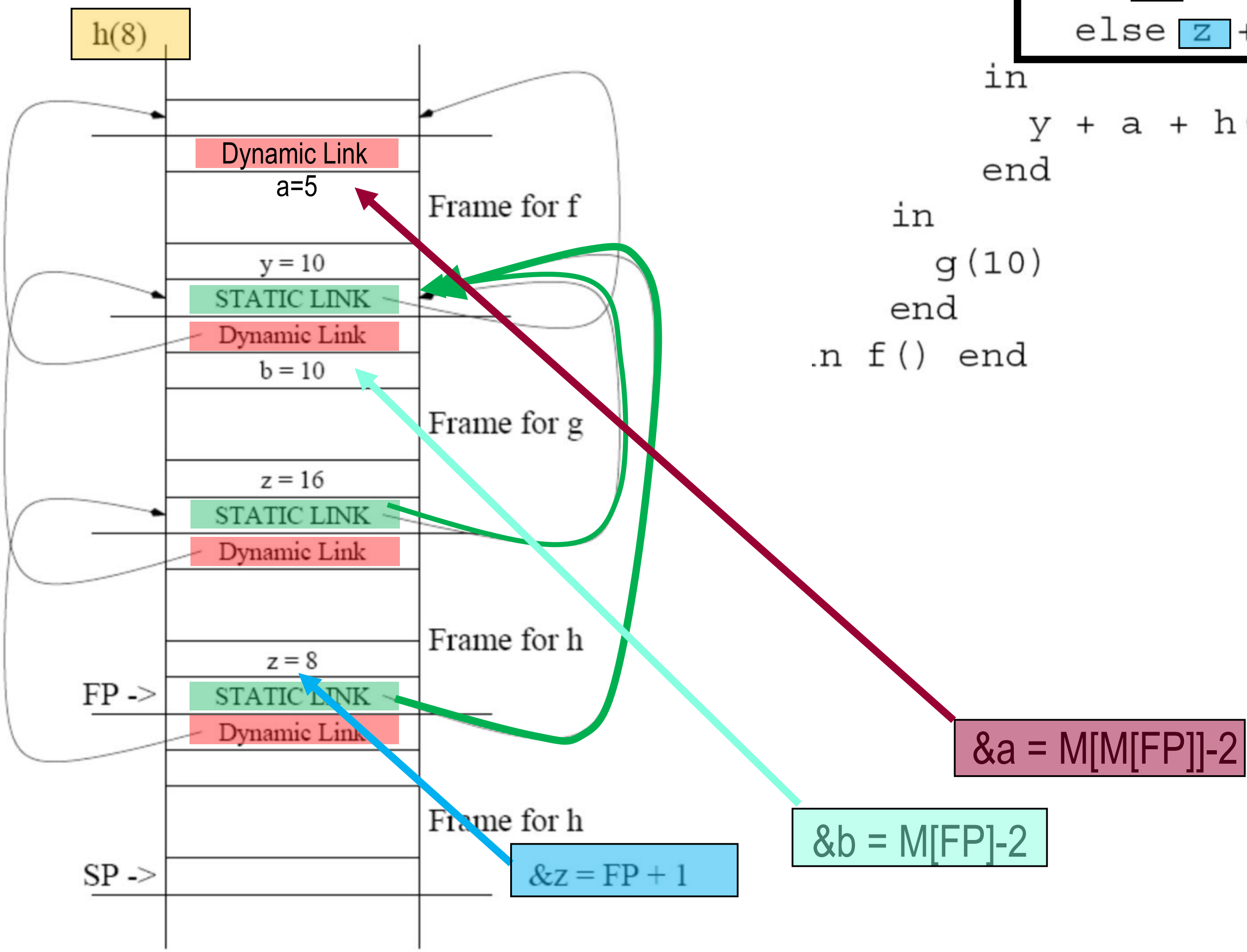
```
let
  function f():int = let
      var a:=5
      function g(y:int):int = let
          var b:=10
          function h(z:int):int =
            if z > 10 then h(z / 2)
            else z + b * a
        in
          y + a + h(16)
        end
    in
      g(10)
    end
in f() end
```



Static Links: recursive call

- Dynamic links point to FP of caller
- Static links point to FP of surrounding function's most recent instance

```
let
  function f():int = let
      var a:=5
      function g(y:int):int = let
          var b:=10
          function h(z:int):int =
            if z > 10 then h(z / 2)
            else z + b * a
        in
          y + a + h(16)
        end
    in
      g(10)
    end
.n f() end
```



Static Links

- Need a chain of indirect memory references for each variable access.
- Number of indirect references = difference in nesting depth between variable declaration function and use function.
- dynamic link still needed to restore caller's FP during function return
- offsets on slides 22-24 need to be modified by +/- 1 to account for the extra slot used by the static link.