Computer Organisation & Program Execution 2021

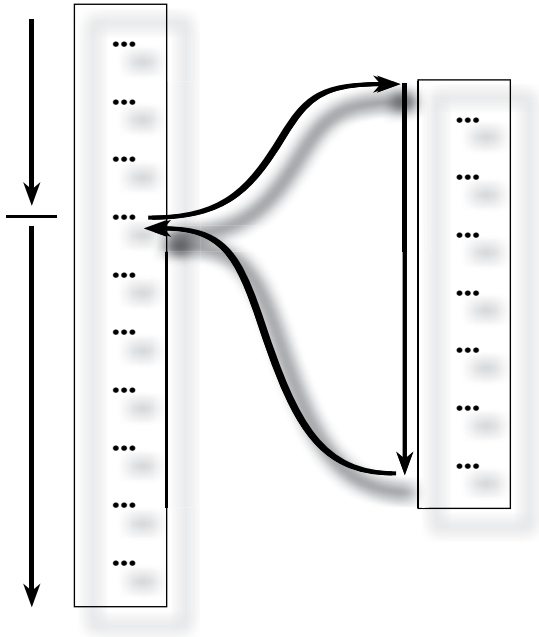
3

Functions

Uwe R. Zimmer - The Australian National University

References for this chapter

[Patterson17]
David A. Patterson & John L. Hennessy
Computer Organization and Design – The Hardware/Software Interface
Chapter 2 "Instructions: Language of the Computer"
ARM edition, Morgan Kaufmann 2017



(Greatness from …) Small beginnings

```haskell
plus_1 :: (Num a) => a -> a
plus_1 x = x + 1
```

```
int plus1 (int x) {
   return x + 1;
}
```

```
function Plus_1 (x : Integer) return Integer is (x + 1);
```

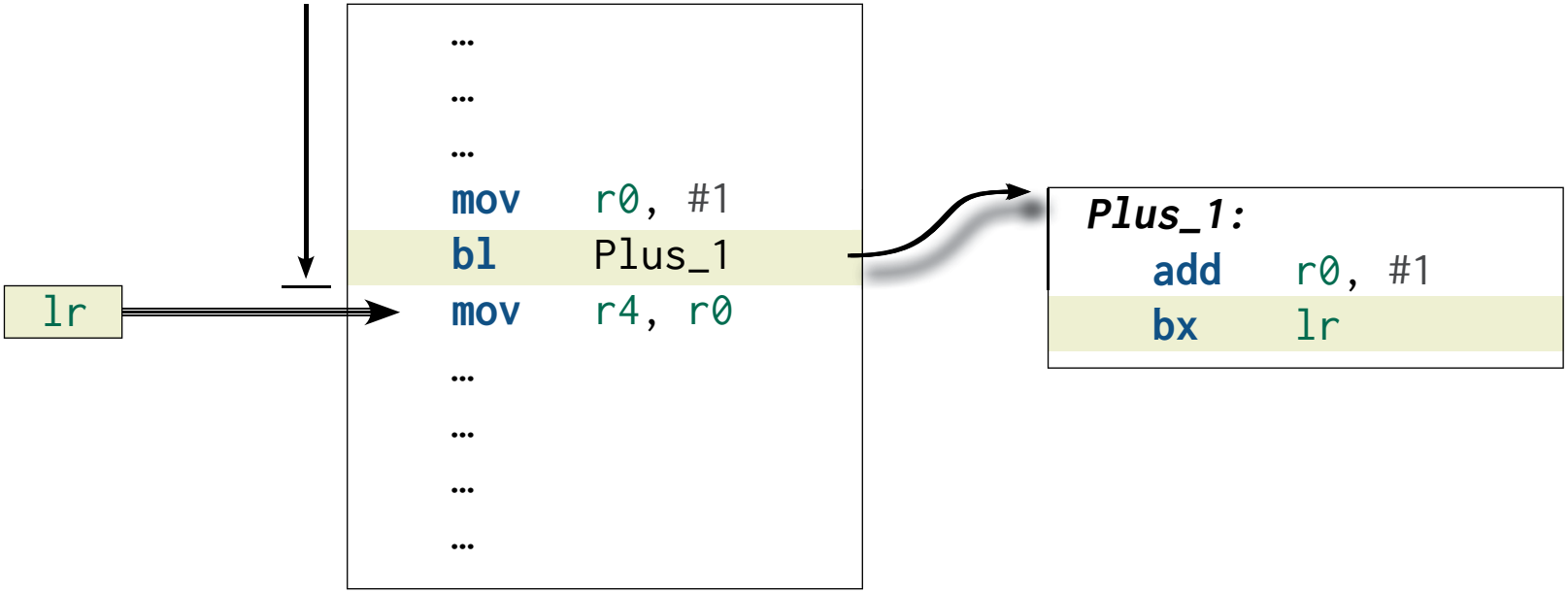
```
def plus1 (x):
   return x + 1;
```

```
pure function plus_1 (x)
   int, intent (in) :: x
   int              :: plus_1
   plus_1 = x + 1;
end function;
```

```
function Plus_1 (x : integer) : integer;
   begin
       Plus_1 := x + 1;
   end;
```

```
    …
    …
    …
    mov     r0, #1
    bl      Plus_1
    mov     r4, r0
    …
    …
    …
    …
```

```
Plus_1:
    add     r0, #1
    bx      lr
```


lr
...
...
...
mov r0, #1
bl Plus_1
mov r4, r0
...
...
...
...
Plus_1:
add r0, #1
bx lr

☞ How is the parameter x passed?

☞ Does it work?

```
...
...
...
mov     r0, #1
bl      Plus_1
mov     r4, r0
...
...
...
...
```

lr

```
Plus_1:
    add     r0, #1
    bx      lr
```

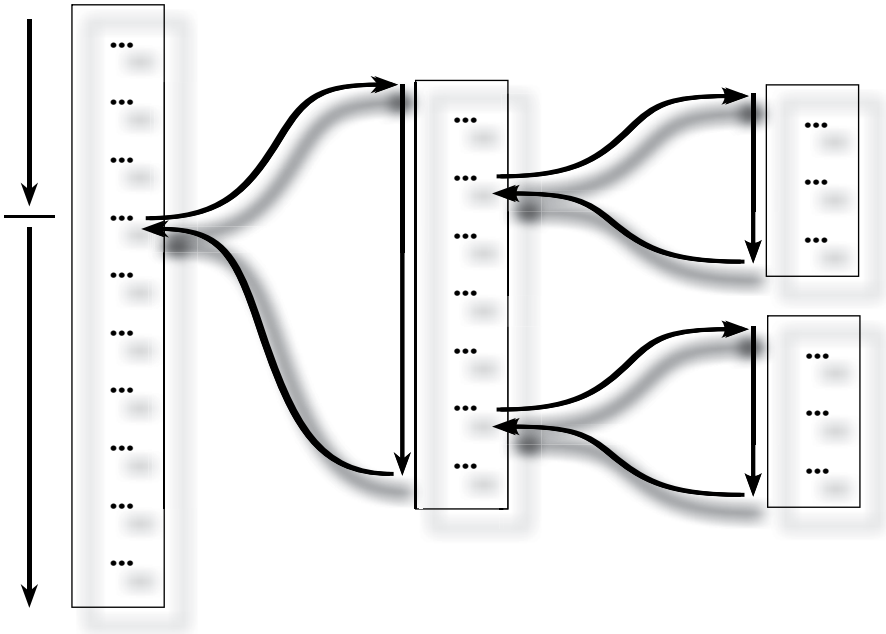
☞ Could it be done differently?

☞ Where do you find the result after the function returns?

```
...
...
...
mov   r0, #1
add   r0, #1
mov   r4, r0
...
...
...
...
```

☞ This is called **inlining**

☞ Can/should this always be done?



(Greatness from …) Small beginnings

```
plus_2 :: (Num a) => a -> a
plus_2 x = plus_1 $ plus_1 x
```

```
int plus2 (int x) {
   return plus1 (plus1 (x));
}
```

```
function Plus_2 (x : Integer) return Integer is (Plus_1 (Plus_1 (x)));
```

```
def plus2 (x):
   return plus1 (plus1 (x));
```

```
pure function plus_2 (x)
   int, intent (in) :: x
   int              :: plus_2
   plus_2 = plus_1 (plus_1 (x));
end function;
```

```
function Plus_2 (x : integer) : integer;
   begin
       Plus_2 := Plus_1 (Plus_1 (x));
   end;
```

```
…
…
…
mov     r0, #1
bl      Plus_2
mov     r3, r0
…
…
…
…
```

```
Plus_2:
    …
bl Plus_1
bl Plus_1
    …
    bx     lr
```

```
Plus_1:
    add     r0, #1
    bx      lr
```

```
bx      lr
```

is used three times

☞ What is the value of lr in each case?

```
…
…
…
mov     r0, #1
bl      Plus_2
mov     r3, r0
…
…
…
…
```

```
Plus_2:
    str     lr, [sp, #-4]!
    bl Plus_1
    bl Plus_1
    ldr     lr, [sp], #4
    bx      lr
```

```
Plus_1:
    add     r0, #1
    bx      lr
```

```
bx      lr
```

is used three times

☞ What is the value of lr in each case?

```
...
...
...
mov   r0, #1
add   r0, #2
mov   r3, r0
...
...
...
...
```

... we need an example, where a compiler will not just remove all our code!

(Greatness from …) Small beginnings

```
fib_fact :: (Num a) => a -> a
fib_fact x = (fib x) + (fact x)
```

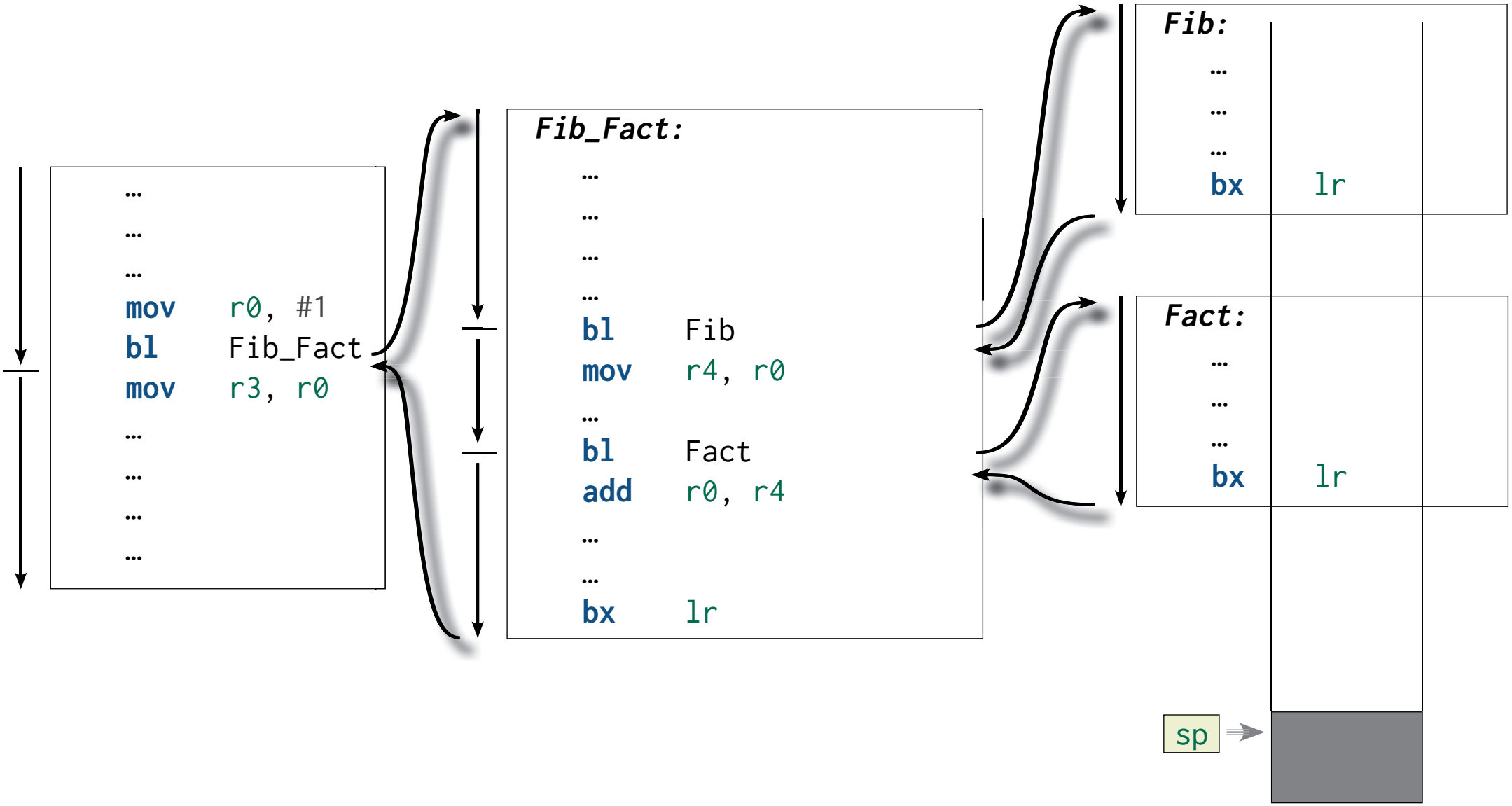
```
unsigned int fibFact (unsigned int x) {
   return fib (x) + fact (x);
}
```

```
function Fib_Fact (x : Natural) return Natural is (Fib (x) + Fact (x));
```

```
def fibFact (x):
   return fib (x) + fact (x);
```

```
pure function fib_fact (x)
   int, intent (in) :: x
   int              :: fib_fact
   fib_fact = fib (x) + fact (x);
end function;
```

```
function Fib_Fact (x : cardinal) : cardinal;
   begin
       Fib_Fact := Fib (x) + Fact (x);
   end;
```


…
…
…
mov r0, #1
bl Fib_Fact
mov r3, r0
…
…
…
…
Fib_Fact:
…
…
…
…
bl Fib
mov r4, r0
…
bl Fact
add r0, r4
…
…
bx lr
Fib:
…
…
…
bx lr
Fact:
…
…
…
bx lr
sp

```
…
…
…
mov     r0, #1
bl      Fib_Fact
mov     r3, r0
…
…
…
…
```

```
Fib_Fact:
…
…
…
…
bl      Fib
mov     r4, r0
…
bl      Fact
add     r0, r4
…
…
bx      lr
```

```
Fib:
…
…
…
bx      lr
```

```
Fact:
…
…
…
bx      lr
```

```
bx      lr
```

… where does this lead us?

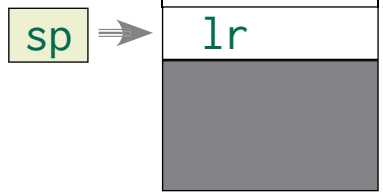
sp

```
…
…
…
mov     r0, #1
bl      Fib_Fact
mov     r3, r0
…
…
…
…
```

```
Fib_Fact:
    str     lr, [sp, #-4]!
    …
    …
    …
    bl      Fib
    mov     r4, r0
    …
    bl      Fact
    add     r0, r4
    …
    ldr     lr, [sp], #4
    bx      lr
```

```
Fib:
    …
    …
    …
    bx      lr
```

```
Fact:
    …
    …
    …
    bx      lr
```



```
…
…
…
mov     r0, #1
bl      Fib_Fact
mov     r3, r0
…
…
…
…
```

```
Fib_Fact:
    str     lr, [sp, #-4]!
    …
    …
    …
    bl      Fib
    mov     r4, r0
    …
    bl      Fact
    add     r0, r4
    …
    ldr     lr, [sp], #4
    bx      lr
```

```
Fib:
    …
    …
    …
    bx      lr
```

```
Fact:
    …
    …
    …
    bx      lr
```

sp → lr

r4

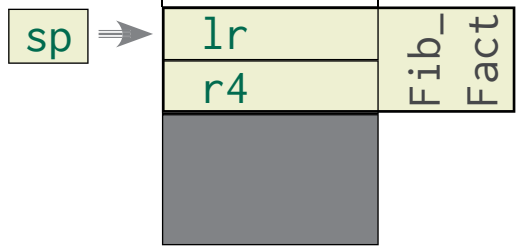
… what if this was holding some information at the time when we were called?

```
…
…
…
mov     r0, #1
bl      Fib_Fact
mov     r3, r0
…
…
…
…
```

```
Fib_Fact:
    stmdb sp!, {r4, lr}
    …
    …
    …
    bl     Fib
    mov    r4, r0
    …
    bl     Fact
    add    r0, r4
    …
    ldmia sp!, {r4, lr}
    bx     lr
```

```
Fib:
    …
    …
    …
    bx     lr
```

```
Fact:
    …
    …
    …
    bx     lr
```

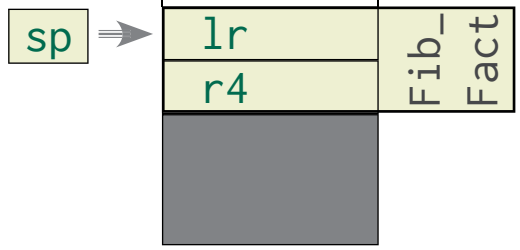


```
…
…
…
mov     r0, #1
bl      Fib_Fact
mov     r3, r0
…
…
…
…
```

```
Fib_Fact:
    stmdb sp!, {r4, lr}
    …
    …
    …
    bl    Fib
    mov   r4, r0
    …
    bl    Fact
    add   r0, r4
    …
    ldmia sp!, {r4, lr}
    bx    lr
```

```
Fib:
    …
    …
    …
    bx    lr
```

```
Fact:
    …
    …
    …
    bx    lr
```



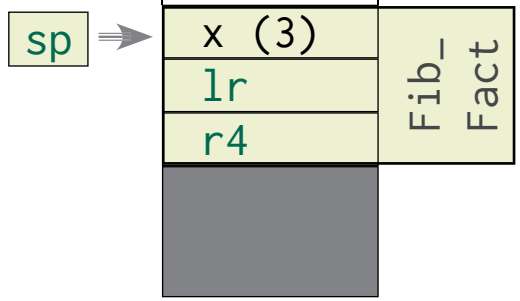
What happens to our parameter x (1) during the function?

```
…
…
…
mov     r0, #1
bl      Fib_Fact
mov     r3, r0
…
…
…
…
```

```
Fib_Fact:
    stmdb sp!, {r4, lr}
    …
    sub   sp, #4
    str   r0, [sp]
    bl    Fib
    mov   r4, r0
    ldr   r0, [sp]
    bl    Fact
    add   r0, r4
    add   sp, #4
    ldmia sp!, {r4, lr}
    bx    lr
```

```
Fib:
    …
    …
    …
    bx    lr
```

```
Fact:
    …
    …
    …
    bx    lr
```

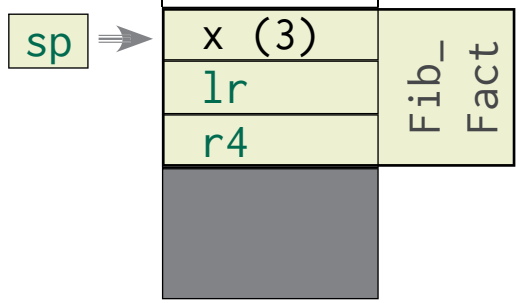


```
…
…
…
mov     r0, #1
bl      Fib_Fact
mov     r3, r0
…
…
…
…
```

```
Fib_Fact:
    stmdb sp!, {r4, lr}
    …
    sub   sp, #4
    str   r0, [sp]
    bl    Fib
    mov   r4, r0
    ldr   r0, [sp]
    bl    Fact
    add   r0, r4
    add   sp, #4
    ldmia sp!, {r4, lr}
    bx    lr
```

```
Fib:
    …
    …
    …
    bx    lr
```

```
Fact:
    …
    …
    …
    bx    lr
```



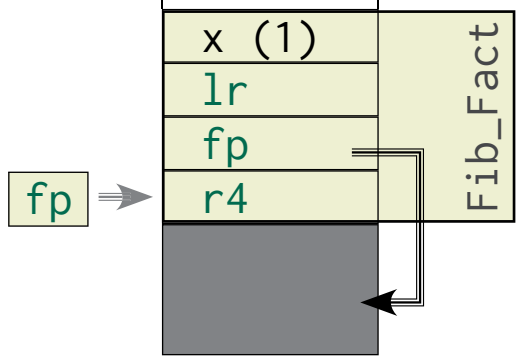
While addressing via the sp is possible, it may also be complex to keep track of, as the sp may change further.

```
…
…
…
mov     r0, #1
bl      Fib_Fact
mov     r3, r0
…
…
…
…
```

```
Fib_Fact:
    stmdb sp!, {r4, fp, lr}
    add   fp, sp, #8
    sub   sp, #4
    str   r0, [fp, #-12]
    bl    Fib
    mov   r4, r0
    ldr   r0, [fp, #-12]
    bl    Fact
    add   r0, r4
    add   sp, #4
    ldmia sp!, {r4, fp, lr}
    bx    lr
```

```
Fib:
    …
    …
    …
    bx    lr
```

```
Fact:
    …
    …
    …
    bx    lr
```



Keeping a reference to the start of the **Stack Frame** for this function (with the frame-pointer fp) makes things neater and enables structured access to the dynamic context.

Recursive

```
unsigned int fib (unsigned int x) {
  switch (x) {
    case 0  : return 0;
    case 1  : return 1;
    default : return fib (x - 1) + fib (x - 2);
} }
```

```
function Fib (x : Natural) return Natural is
  (case x is
     when 0      => 0,
     when 1      => 1,
     when others => Fib (x - 1) + Fib (x - 2));
```

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

Will our stack handling work for recursive functions?

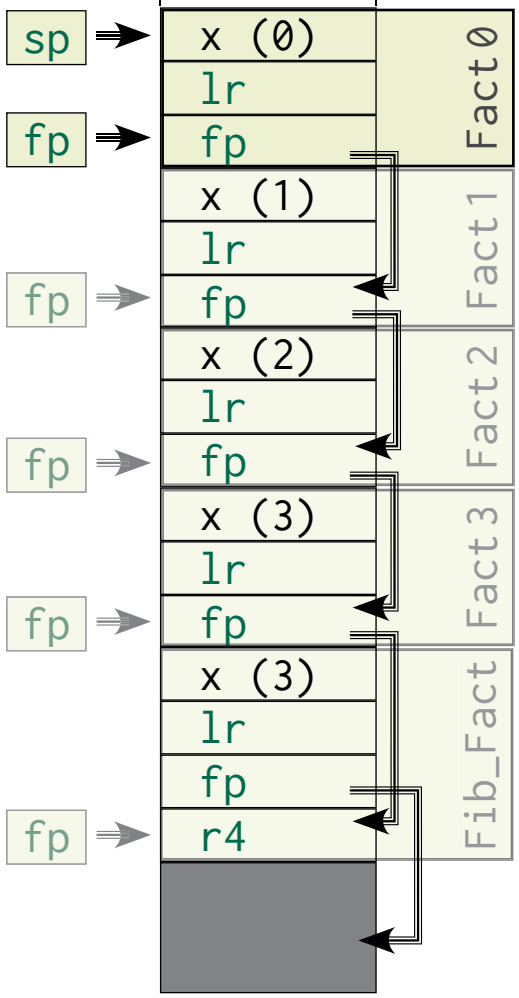
```
function Fact (x : Natural) return Positive is
  (if x = 0 then 1
            else x * Fact (x - 1));
```

☞ Is Fact **reentrant**?

☞ How high do we stack?

```
Fib_Fact:
    stmdb sp!, {r4, fp, lr}
    add   fp, sp, #8
    sub   sp, #4
    str   r0, [fp, #-12]
    bl    Fib
    mov   r4, r0
    ldr   r0, [fp, #-12]
    bl    Fact
    add   r0, r0, r4
    add   sp, #4
    ldmia sp!, {r4, fp, lr}
    bx    lr
```

```
Fact:
    stmdb sp!, {fp, lr}
    add   fp, sp, #4
    sub   sp, #4
    str   r0, [fp, #-8]
    cmp   r0, #0
    bne   Case_Others
    mov   r0, #1
    b     End_Fact
Case_Others:
    sub   r0, #1
    bl    Fact
    mov   r1, r0
    ldr   r0, [fp, #-8]
    mul   r0, r1
End_Fact:
    add   sp, #4
    ldmia sp!, {fp, lr}
    bx    lr
```



☞ Where is the last lr stored?

```
function Fact (x : Natural) return Positive is
  (if x = 0 then 1
            else x * Fact (x - 1));
```

☞ A compiler will likely replace such a recursion!

```
function Fact (x : Natural) return Positive is
  Fac : Positive := 1;
begin
  for i in 1 .. x loop
    Fac := Fac * i;
  end loop;
  return Fac;
end Fact;
```

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

☞ A compiler will likely replace such a recursion!

```
unsigned int fact (unsigned int x) {
  int fac = 1;
  for (i = 1, i <= x, i++) {
    fac = fac * i;
  }
  return fac;
}
```

Functions

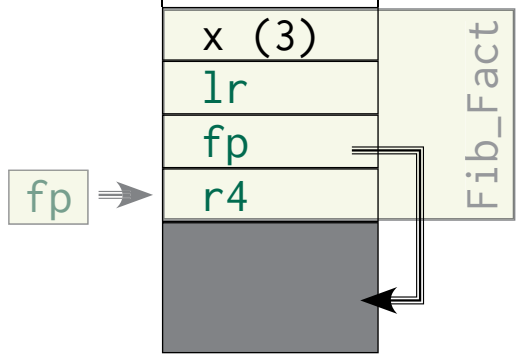
```
Fib_Fact:
    stmdb sp!, {r4, fp, lr}
    add   fp, sp, #8
    sub   sp, #4
    str   r0, [fp, #-12]
    bl    Fib
    mov   r4, r0
    ldr   r0, [fp, #-12]
    bl    Fact
    add   r0, r0, r4
    add   sp, #4
    ldmia sp!, {r4, fp, lr}
    bx    lr
```

```
Fib:
    …
    …
    …
    bx    lr
```

```
Fact:
    adds  r3, r0, #0
    mov   r0, #1
    beq   End_Fact
Fact_Loop:
    mul   r0, r3
    subs  r3, #1
    bne   Fact_Loop
End_Fact:
    bx    lr
```

☞ Complexity? Runtimes?

☞ Is Fact **reentrant**?



Besides all the inlining, unrolling, flattening, etc. :

☞ Stack operations are still vital for any non-trivial program.

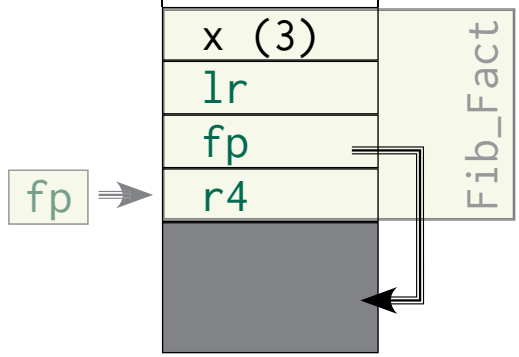
```
Fib_Fact:
    stmdb sp!, {r4, fp, lr}
    add   fp, sp, #8
    sub   sp, #4
    str   r0, [fp, #-12]
    bl    Fib
    mov   r4, r0
    ldr   r0, [fp, #-12]
    bl    Fact
    add   r0, r0, r4
    add   sp, #4
    ldmia sp!, {r4, fp, lr}
    bx    lr
```

```
Fib:
    ...
    ...
    bx    lr
```

☞ Why do we save r4?

```
Fact:
    adds  r3, r0, #0
    mov   r0, #1
    beq   End_Fact
Fact_Loop:
    mul   r0, r3
    subs  r3, #1
    bne   Fact_Loop
End_Fact:
    bx    lr
```

☞ But we don't save r3?



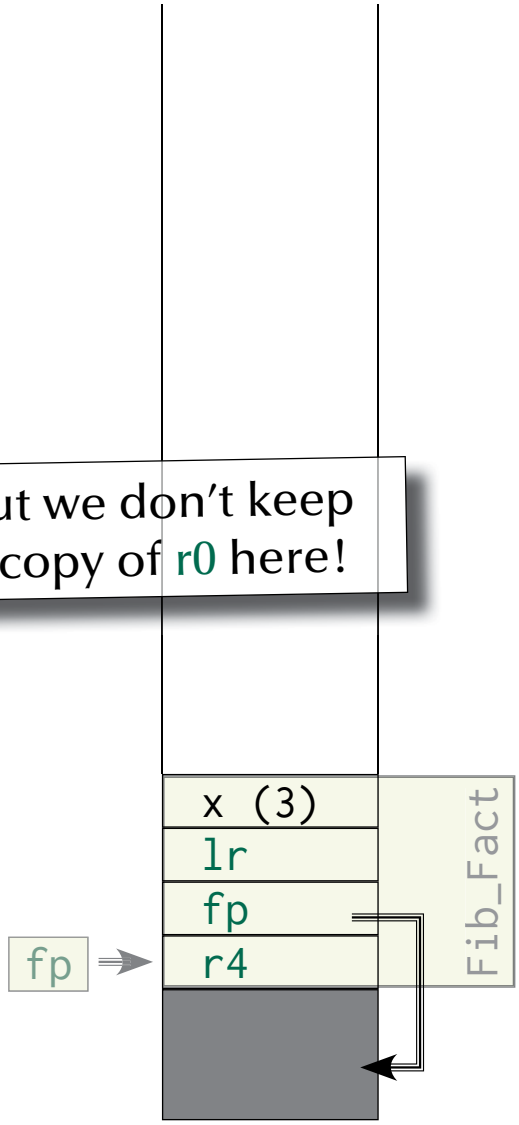
```
Fib_Fact:
    stmdb sp!, {r4, fp, lr}
    add   fp, sp, #8
    sub   sp, #4
    str   r0, [fp, #-12]
    bl    Fib
    mov   r4, r0
    ldr   r0, [fp, #-12]
    bl    Fact
    add   r0, r0, r4
    add   sp, #4
    ldmia sp!, {r4, fp, lr}
    bx    lr
```

```
Fib:
    …
    …
    …
    bx    lr
```

☞ We keep a copy of r0 here.

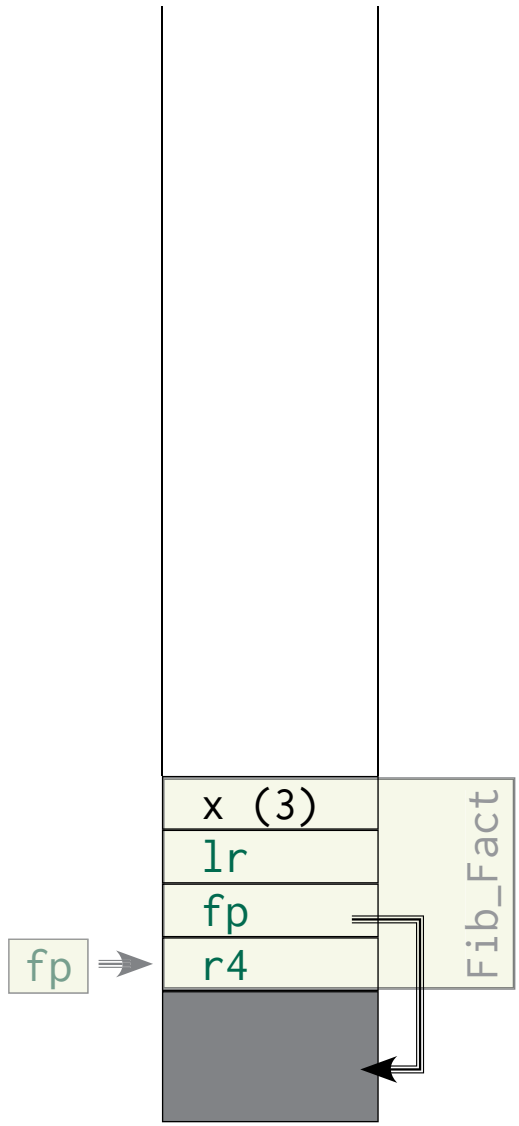
```
Fact:
    adds  r3, r0, #0
    mov   r0, #1
    beq   End_Fact
Fact_Loop:
    mul   r0, r3
    subs  r3, #1
    bne   Fact_Loop
End_Fact:
    bx    lr
```

☞ But we don't keep a copy of r0 here!



```
Fib_Fact:
    stmdb sp!, {r4, fp, lr}
    add   fp, sp, #8
    sub   sp, #4
    str   r0, [fp, #-12]
    bl    Fib
    mov   r4, r0
    ldr   r0, [fp, #-12]
    bl    Fact
    add   r0, r0, r4
    add   sp, #4
    ldmia sp!, {r4, fp, lr}
    bx    lr
```

```
Fib:
    stmdb sp!, {r4, fp, lr}
    add   fp, sp, #8
    sub   sp, #4
    str   r0, [fp, #-12]
    cmp   r0, #0
    beq   End_Fib
    cmp   r0, #1
    beq   End_Fib
    sub   r0, r0, #1
    bl    Fib
    mov   r4, r0
    ldr   r0, [fp, #-12]
    sub   r0, r0, #2
    bl    Fib
    add   r0, r4, r0
End_Fib:
    add   sp, #4
    ldmia sp!, {r4, fp, lr}
    bx    lr
```



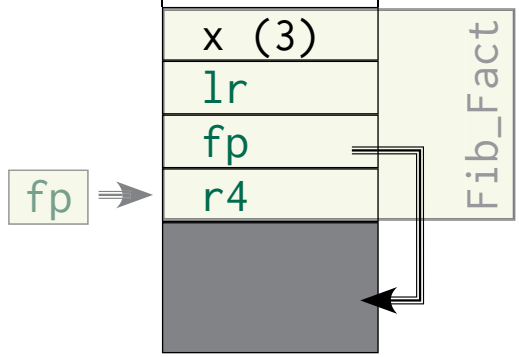
```
Fib_Fact:
    stmdb sp!, {r4, fp, lr}
    add   fp, sp, #8
    sub   sp, #4
    str   r0, [fp, #-12]
    bl    Fib
    mov   r4, r0
    ldr   r0, [fp, #-12]
    bl    Fact
    add   r0, r0, r4
    add   sp, #4
    ldmia sp!, {r4, fp, lr}
    bx    lr
```

☞ There could be **two** further Fib-calls for each call to Fib …

```
Fib:
    stmdb sp!, {r4, fp, lr}
    add   fp, sp, #8
    sub   sp, #4
    str   r0, [fp, #-12]
    cmp   r0, #0
    beq   End_Fib
    cmp   r0, #1
    beq   End_Fib
    sub   r0, r0, #1
    bl    Fib
    mov   r4, r0
    ldr   r0, [fp, #-12]
    sub   r0, r0, #2
    bl    Fib
    add   r0, r4, r0
End_Fib:
    add   sp, #4
    ldmia sp!, {r4, fp, lr}
    bx    lr
```

☞ What would be the maximal depth for the stack?

☞ What would the stack look like?



Components / phases of a function call:

- Values (**parameters**) to be passed to a function.
- **Local variables** inside a function.
- Values (**results**) to be returned from a function.

So far we:

☞ … **passed parameter** values in registers (r0 - r3).

☞ … **called the function** (store the return address and jump to the beginning of the function).

☞ … **pushed** the return address, the previous stack frame and used registers (r4 …).

☞ … **created a new stack frame** (and addressed all local variables relative to this).

☞ … **grew the stack** such that it can hold the local variables.

Prologue

☞ … **done the calculations/operations** based on the local variables and scratch registers.

☞ … **passed return values** in registers (r0 - r1).

☞ … **restored the stack pointer** (and thus de-allocated all local variables).

☞ … **popped** the return address, the previous stack frame and used registers (r4 …).

☞ … **jumped back** to the next instruction after the original function call.

Epilogue

☞ … **used the return values** found in r0 - r1.

Components / phases of a function call:

- Values (**parameters**) to be passed to a function.
- **Local variables** inside a function.
- Values (**results**) to be returned from a function.

☞ What happens if the parameters or return values won't fit into registers?

So far we:

☞ … **passed parameter** values in registers (r0 - r3).

☞ … **called the function** (store the return address and jump to the beginning of the function).

☞ … **pushed** the return address, the previous stack frame and used registers (r4 …).

☞ … **created a new stack frame** (and addressed all local variables relative to this).

☞ … **grew the stack** such that it can hold the local variables.

Prologue

☞ … **done the calculations/operations** based on the local variables and scratch registers.

☞ … **passed return values** in registers (r0 - r1).

☞ … **restored the stack pointer** (and thus de-allocated all local variables).

☞ … **popped** the return address, the previous stack frame and used registers (r4 …).

☞ … **jumped back** to the next instruction after the original function call.

Epilogue

☞ … **used the return values** found in r0 - r1.

Conventions

ARM architecture calling practice

- r0-r3 are used for parameters.
- r0-r1 are used for return values.
- r0-r3 are not expected to be intact after a function call … all other registers are expected to be intact!

☞ If those registers do not suffice, additional parameters and results are passed via the stack.

☞ There are also memory alignment constraints.

(Mostly due to memory bus constraints)

☞ Conventions are different in other architectures (e.g. x86, where parameters are generally passed via the stack).

☞ Why are these conventions architecture related at all?

Parameter passing

Call by …

		Access	
		by copy	by reference
Information flow	in	**by value** Parameter becomes a constant inside the function or is copied into a local variable.	**by reference (immutable)** No write access is allowed while the function runs (also from outside the function).
	out	**by result** Calling function expects the parameter value to appear in a specific space at return.	**by reference (mutable, no read)** No read access from inside the function, write access on return.
	in & out	**by value result** Parameter is copied to a local variable and copied back at return.	**by reference (mutable)** Function can read and write at any time. Outside code shall not write.

C

Full control over those three modes.
"by value" parameters are local variables.
"in & out, by reference" syntactically as "by pointer value" parameters.

		Access	
		by copy	by reference
Information flow	in	**by value** Parameter becomes a constant inside the function or is copied into a local variable.	**by reference (immutable)** No write access is allowed while the function runs (also from outside the function).
	out	**by result** Calling function expects the parameter value to appear in a specific space at return.	**by reference (mutable, no read)** No read access from inside the function, write access on return.
	in & out	**by value result** Parameter is copied to a local variable and copied back at return.	**by reference (mutable)** Function can read and write at any time. Outside code shall not write.

Haskell

Only control over information flow – not over access.

		Access	
		by copy	by reference
Information flow	in	**by value** Parameter becomes a constant inside the function or is copied into a local variable.	**by reference (immutable)** No write access is allowed while the function runs (also from outside the function).
	out	**by result** Calling function expects the parameter value to appear in a specific space at return.	**by reference (mutable, no read)** No read access from inside the function, write access on return.
	in & out	**by value result** Parameter is copied to a local variable and copied back on return.	**by reference (mutable)** Function can read and write at any time. Callee should not write.

"in & out" parameters are side-effecting and can therefore not exist in a pure functional language.

Functions

Python

All parameter access is double-indirect (☞ **handles**).

… this has an impact on performance.

		Access	
		by copy	by reference
Information flow	in	**by value** Parameter becomes a constant inside the function or is copied into a local variable.	**by reference (immutable)** No write access is allowed while the function runs (also from outside the function).
	out	**by result** Calling function expects the parameter value to appear in a specific space at return.	**by reference (mutable, no read)** No read access from inside the function, write access on return.
	in & out	**by value result** Parameter is copied to a local variable and copied back at return.	**by reference (mutable)** Function can read and write at any time. Outside code shall not write.

Ada

Limited control over “by value result”.

“by value” parameters are constants.

		Access	
		by copy	by reference
Information flow	in	**by value** Parameter becomes a constant inside the function or is copied into a local variable.	**by reference (immutable)** No write access is allowed while the function runs (also from outside the function).
	out	**by result** Calling function expects the parameter value to appear in a specific space at return.	**by reference (mutable, no read)** No read access from inside the function, write access on return.
	in & out	**by value result** Parameter is copied to a local variable and copied back at return.	**by reference (mutable)** Function can read and write at any time. Outside code shall not write.

Assembly

"By reference" semantics by convention only.

		Access	
		by copy	by reference
Information flow	in	**by value** Parameter becomes a constant inside the function or is copied into a local variable.	**by reference (immutable)** No write access is allowed while the function runs (also from outside the function).
	out	**by result** Calling function expects the parameter value to appear in a specific space at return.	**by reference (mutable, no read)** No read access from inside the function, write access on return.
	in & out	**by value result** Parameter is copied to a local variable and copied back at return.	**by reference (mutable)** Function can read and write at any time. Outside code shall not write.

Parameter passing

Call by name

Pops up in programming languages since the 60's.

… is conceptually a call-by-value, where the value has not been calculated yet.

Technically a reference to a function is passed and the evaluation of this parameter (function) is left to the called function.

Features:

- Values are only evaluated if and when they are needed.
- Values can change during the life-time of a function (in case of side-effecting functions).
- Values can be stored once calculated (in case of side-effect-free functions).

While this is possible to a degree in most programming languages …

(even if there is no specific passing mode, you can still pass a reference to a function)

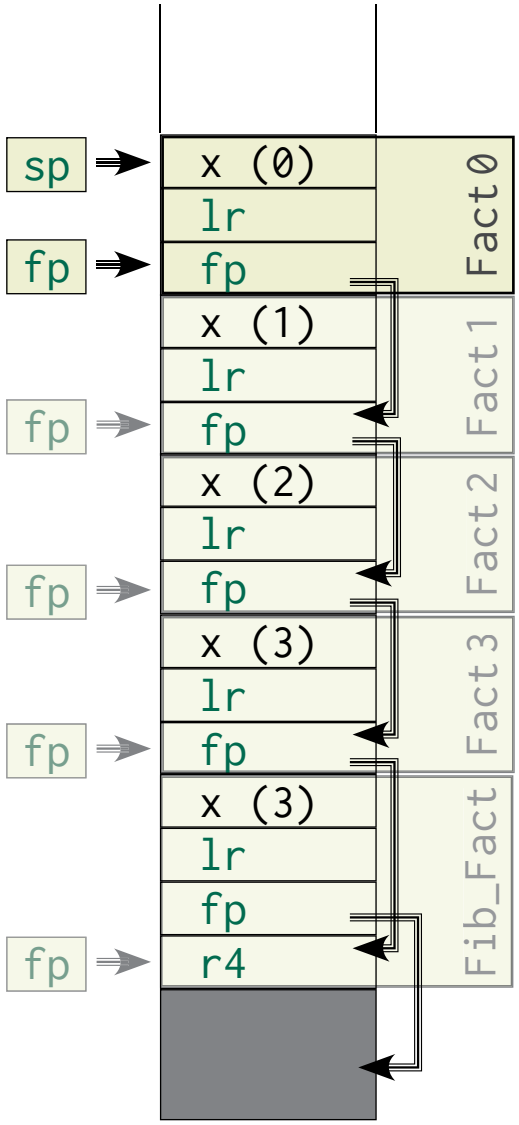
… it is a core concept for functional, lazy evaluation languages, like e.g. Haskell,

and it does find its way back into mainstream languages like C++, .NET languages or Python as **anonymous functions** (sometimes referred to as λ-functions or λ-expressions).

☞ How about using call by reference here?

```
Fib_Fact:
    stmdb sp!, {r4, fp, lr}
    add   fp, sp, #8
    sub   sp, #4
    str   r0, [fp, #-12]
    bl    Fib
    mov   r4, r0
    ldr   r0, [fp, #-12]
    bl    Fact
    add   r0, r0, r4
    add   sp, #4
    ldmia sp!, {r4, fp, lr}
    bx    lr
```

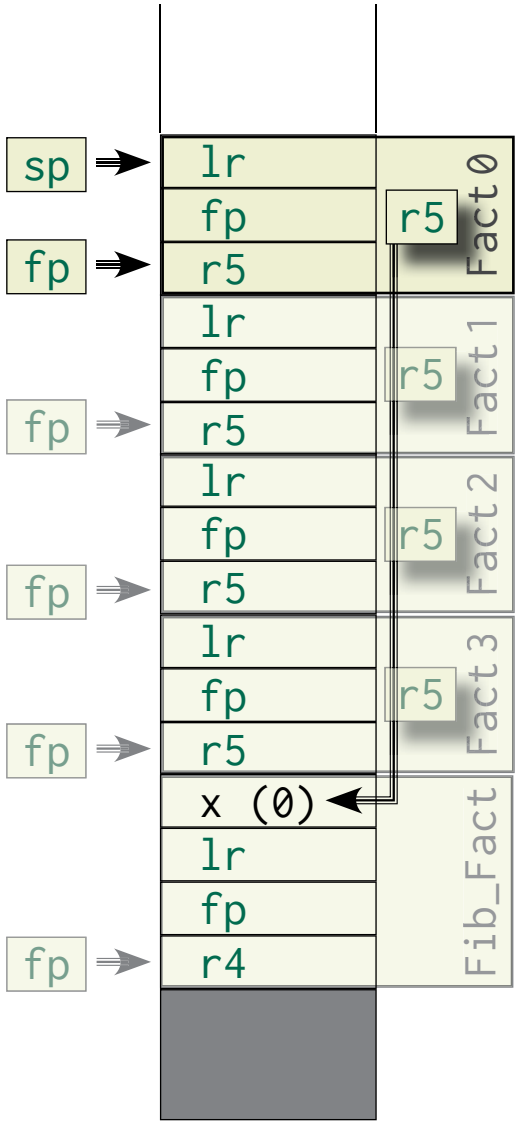
```
Fact:
    stmdb sp!, {fp, lr}
    add   fp, sp, #4
    sub   sp, #4
    str   r0, [fp, #-8]
    cmp   r0, #0
    bne   Case_Others
    mov   r0, #1
    b     End_Fact
Case_Others:
    sub   r0, #1
    bl    Fact
    mov   r1, r0
    ldr   r0, [fp, #-8]
    mul   r0, r1
End_Fact:
    add   sp, #4
    ldmia sp!, {fp, lr}
    bx    lr
```



Functions

```
Fib_Fact:
    stmdb sp!, {r4, fp, lr}
    add   fp, sp, #8
    sub   sp, #4
    str   r0, [fp, #-12]
    bl    Fib
    mov   r4, r0
    sub   r0, fp, #12
    bl    Fact
    add   r0, r0, r4
    add   sp, #4
    ldmia sp!, {r4, fp, lr}
    bx    lr
```

```
Fact:
    stmdb sp!, {r5, fp, lr}
    add   fp, sp, #4
    mov   r5, r0
    ldr   r0, [r5]
    cmp   r0, #0
    bne   Case_Others
    mov   r0, #1
    b     End_Fact
Case_Others:
    sub   r0, #1
    str   r0, [r5]
    mov   r0, r5
    bl    Fact
    mov   r1, r0
    ldr   r0, [r5]
    mul   r0, r1
End_Fact:
    mov   sp, fp
    ldmia sp!, {r5, fp, lr}
    bx    lr
```

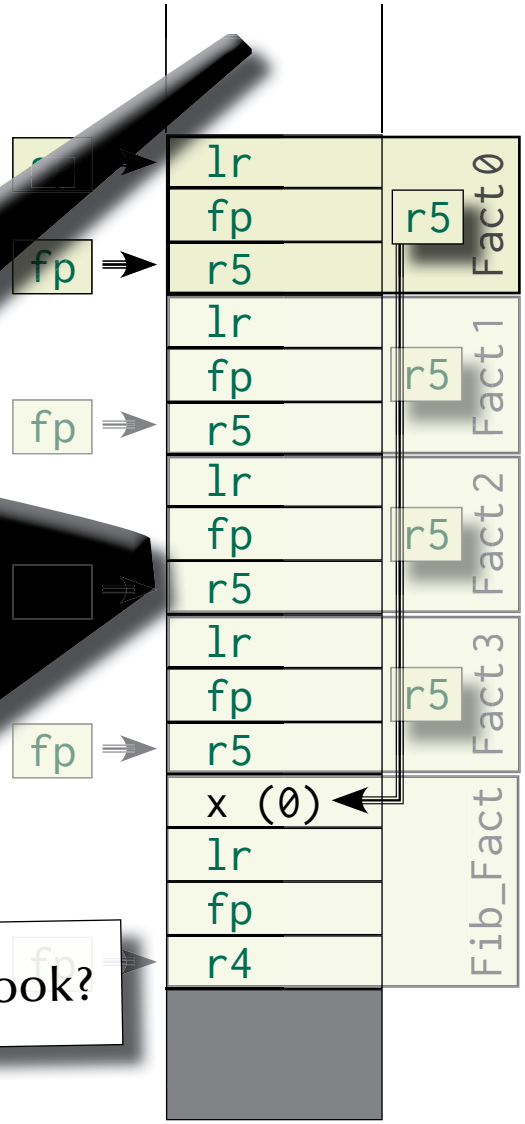


r5 has been nominated to hold the reference to x inside Fact

Functions

```
Fib_Fact:
    stmdb sp!, {r4, fp, lr}
    add   fp, sp, #8
    sub   sp, #4
    str   r0, [fp, #-12]
    bl    Fib
    mov   r4, r0
    sub   r0, fp, #12
    bl    Fact
    add   r0, r0, r4
    add   sp, #4
    ldmia sp!, {r4, fp, lr}
    bx    lr
```

```
Fact:
    stmdb sp!, {r5, fp, lr}
    add   fp, sp, #4
    mov   r5, r0
    ldr   r0, [r5]
    cmp   r0, #0
    bne   Case_Others
    mov   r0, #1
    b     End_Fact
Case_Others:
    sub   r0, [illegible]
    str   r0, [r5]
    mov   r0, r5
    bl    Fact
    mov   r1, r0
    ldr   r0, [r5]
    mul   r0, r[illegible]
End_Fact:
    mov   [illegible]
    ldmia sp!, {r5, fp, lr}
    bx    lr
```



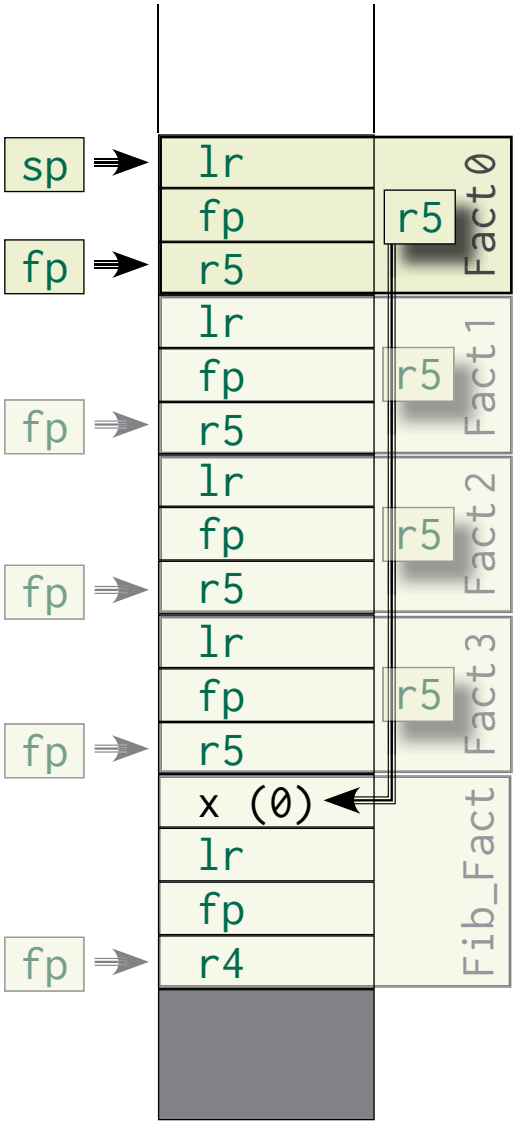
☞ We turned Fact into the constant 0.

☞ What did we overlook?

```
Fib_Fact:
    stmdb sp!, {r4, fp, lr}
    add   fp, sp, #8
    sub   sp, #4
    str   r0, [fp, #-12]
    bl    Fib
    mov   r4, r0
    sub   r0, fp, #12
    bl    Fact
    add   r0, r0, r4
    add   sp, #4
    ldmia sp!, {r4, fp, lr}
    bx    lr
```

What is the value of x during one execution of Fact?

```
Fact:
    stmdb sp!, {r5, fp, lr}
    add   fp, sp, #4
    mov   r5, r0
    ldr   r0, [r5]
    cmp   r0, #0
    bne   Case_Others
    mov   r0, #1
    b     End_Fact
Case_Others:
    sub   r0, #1
    str   r0, [r5]
    mov   r0, r5
    bl    Fact
    mov   r1, r0
    ldr   r0, [r5]
    mul   r0, r1
End_Fact:
    mov   sp, fp
    ldmia sp!, {r5, fp, lr}
    bx    lr
```



Parameter passing

Call by …

		Access	
		by copy	by reference
Information flow	in	**by value** Parameter becomes a constant inside the function or is copied into a local variable.	**by reference (immutable)** No write access is allowed while the function runs (also from outside the function).
	out	**by result** Calling function expects the parameter value to appear in a specific space at return.	**by reference (mutable, no read)** No read access from inside the function, write access on return.
	in & out	**by value result** Parameter is copied to a local variable and copied back at return.	**by reference (mutable)** Function can read and write at any time. Outside code shall not write.

We should have used either of those modes

Yet we used this mode

When to use what?

		Access	
		by copy	by reference
Information flow	in	**by value** Parameter becomes a constant inside the function or is copied into a local variable.	**by reference (immutable)** No write access is allowed while the function runs (also from outside the function).
	out	**by result** Calling function expects the parameter value to appear in a specific space at return.	**by reference (mutable, no read)** No read access from inside the function, write access on return.
	in & out	**by value result** Parameter is copied to a local variable and copied back at return.	**by reference (mutable)** Function can read and write at any time. Outside code shall not write.

One-way and by-copy

Those are side-effect-free and hence the resulting scenarios are easy to analyse.

Copying large data structures might be time consuming or infeasible.

Values can be passed in registers.

		Access	
		by copy	by reference
Information flow	in	**by value** Parameter becomes a constant inside the function or is copied into a local variable.	**by reference (immutable)** No write access is allowed while the function runs (also from outside the function).
	out	**by result** Calling function expects the parameter value to appear in a specific space at return.	**by reference (mutable, no read)** No read access from inside the function, write access on return.
	in & out	**by value result** Parameter is copied to a local variable and copied back at return.	**by reference (mutable)** Function can read and write at any time. Outside code shall not write.

Two-way and by-copy

Still side-effect-free within the function (but not on the outside).

Potentially more convenient as memory space can be reused.

Values can be passed in registers.

		Access	
		by copy	by reference
Information flow	in	**by value** Parameter becomes a constant inside the function or is copied into a local variable.	**by reference (immutable)** No write access is allowed while the function runs (also from outside the function).
	out	**by result** Calling function expects the parameter value to appear in a specific space at return.	**by reference (mutable, no read)** No read access from inside the function, write access on return.
	in & out	**by value result** Parameter is copied to a local variable and copied back at return.	**by reference (mutable)** Function can read and write at any time. Outside code shall not write.

Two-way and by-reference

Side-effecting and particular care is required as multiple entities could write on this.

No data has to be replicated.

Values have to be passed in memory.

		Access	
		by copy	by reference
Information flow	in	**by value** Parameter becomes a constant inside the function or is copied into a local variable.	**by reference (immutable)** No write access is allowed while the function runs (also from outside the function).
	out	**by result** Calling function expects the parameter value to appear in a specific space at return.	**by reference (mutable, no read)** No read access from inside the function, write access on return.
	in & out	**by value result** Parameter is copied to a local variable and copied back at return.	**by reference (mutable)** Function can read and write at any time. Outside code shall not write.

One-way-Out and by-reference

Side-effect-free, if new memory is allocated on return
– cannot be enforced on assembly level (requires compiler).

Values have to be passed in memory.

		Access	
		by copy	by reference
Information flow	in	**by value** Parameter becomes a constant inside the function or is copied into a local variable.	**by reference (immutable)** No write access is allowed while the function runs (also from outside the function).
	out	**by result** Calling function expects the parameter value to appear in a specific space at return.	**by reference (mutable, no read)** No read access from inside the function, write access on return.
	in & out	**by value result** Parameter is copied to a local variable and copied back at return.	**by reference (mutable)** Function can read and write at any time. Outside code shall not write.

One-way-In and by-reference

Side-effect-free – cannot be enforced on assembly level (requires compiler).

No data has to be replicated.

Values have to be passed in memory.

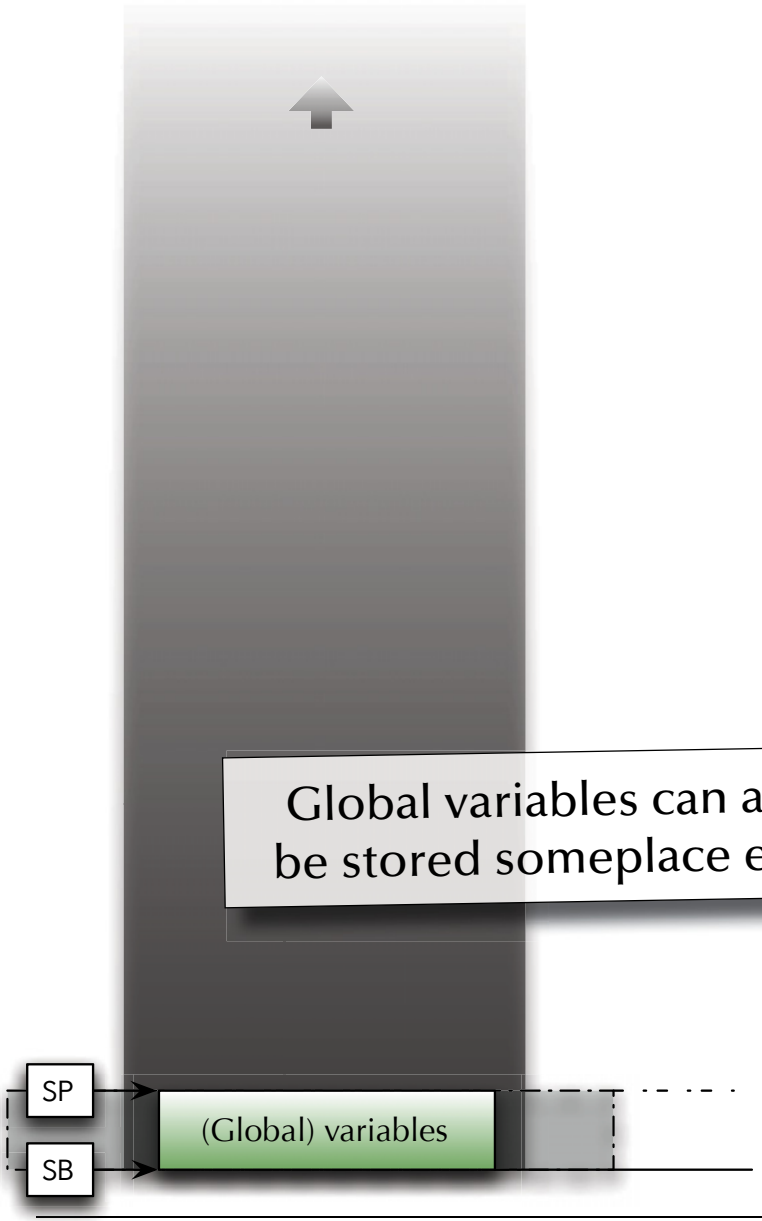
		Access	
		by copy	by reference
Information flow	in	**by value** Parameter becomes a constant inside the function or is copied into a local variable.	**by reference (immutable)** No write access is allowed while the function runs (also from outside the function).
	out	**by result** Calling function expects the parameter value to appear in a specific space at return.	**by reference (mutable, no read)** No read access from inside the function, write access on return.
	in & out	**by value result** Parameter is copied to a local variable and copied back at return.	**by reference (mutable)** Function can read and write at any time. Outside code shall not write.

Generic Stack-Frame

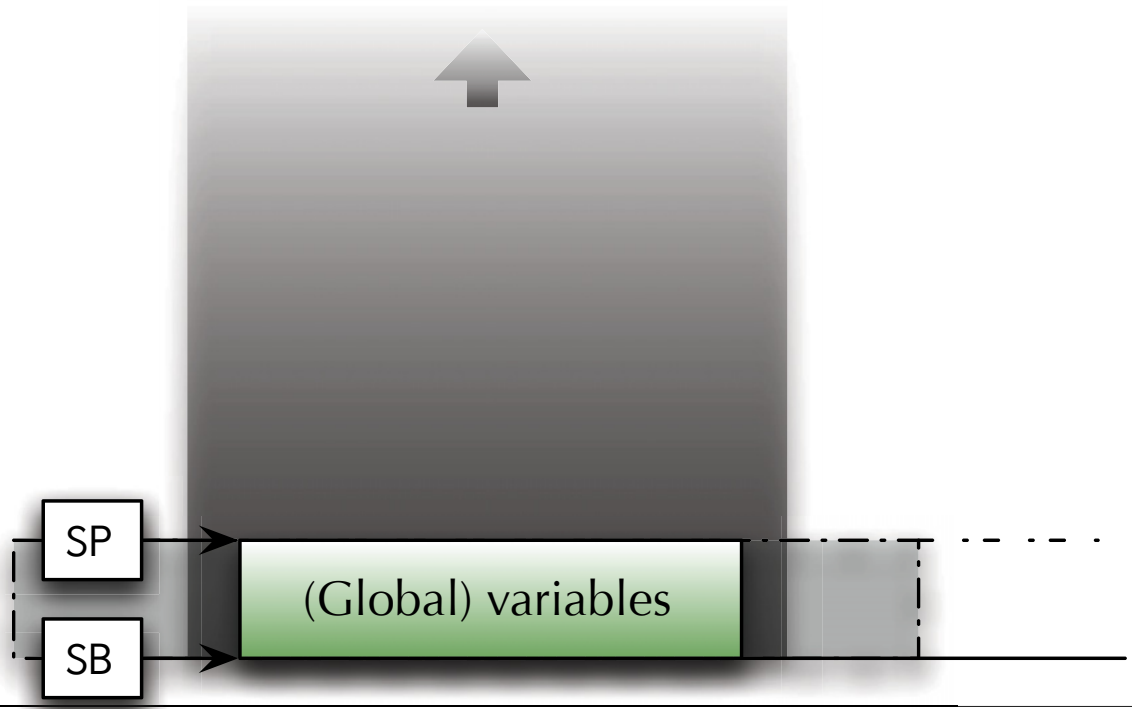
Let there be some (global) data on the stack.

Stack-Base (SB) is a static address,
always pointing to the … well.

Stack-Pointer (SP) points to the current top of the stack.



Global variables can also
be stored someplace else.



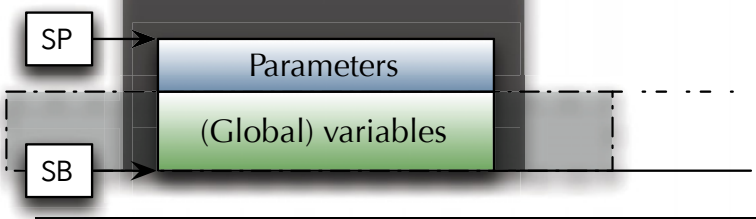
Generic Stack-Frame

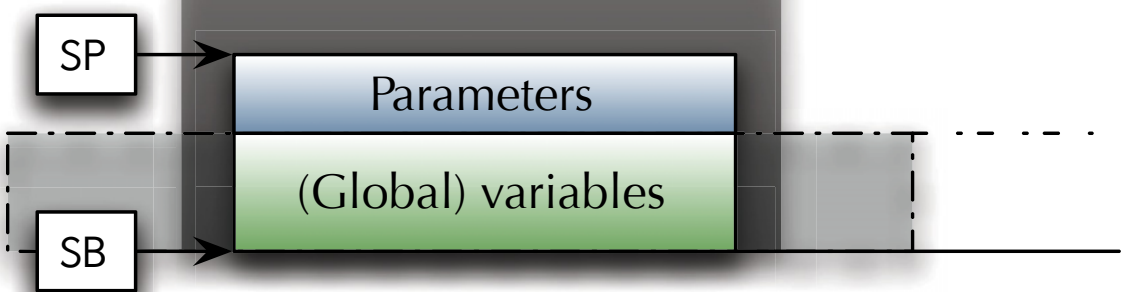
The current code prepared to call a function:

☞ Push parameters on the stack.

Works for any data size (unless the stack overflows) and parameter passing mode.

Types, storage structures and passing modes have to be agreed upon between caller and callee.





Generic Stack-Frame

The current code prepared to call a function:

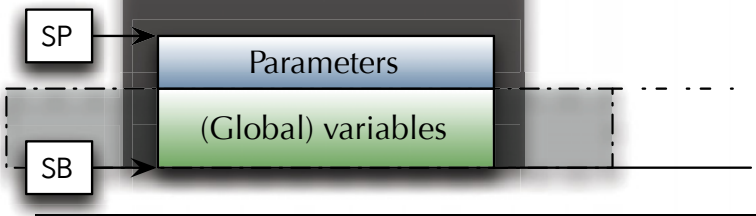
☞ Push parameters on the stack.

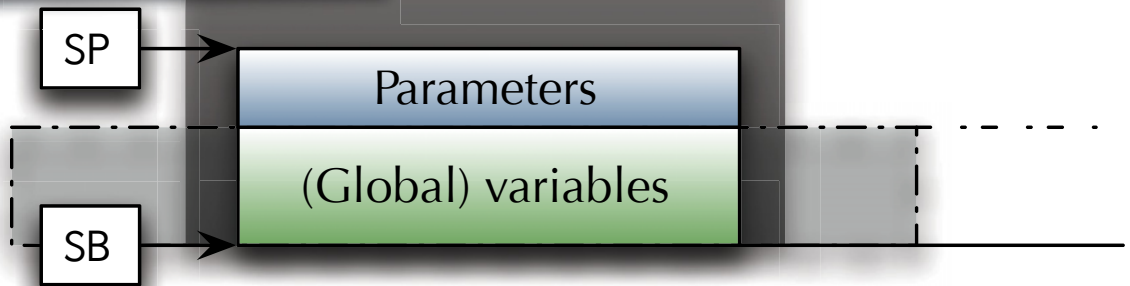
Works for any data size (unless the stack overflows) and parameter passing mode.

Types, storage structures and passing modes have to be agreed upon between caller and callee.

☞ Solved if it's the same language, compiler and program.

☞ If the languages or the compilers are different, then standards will be required.



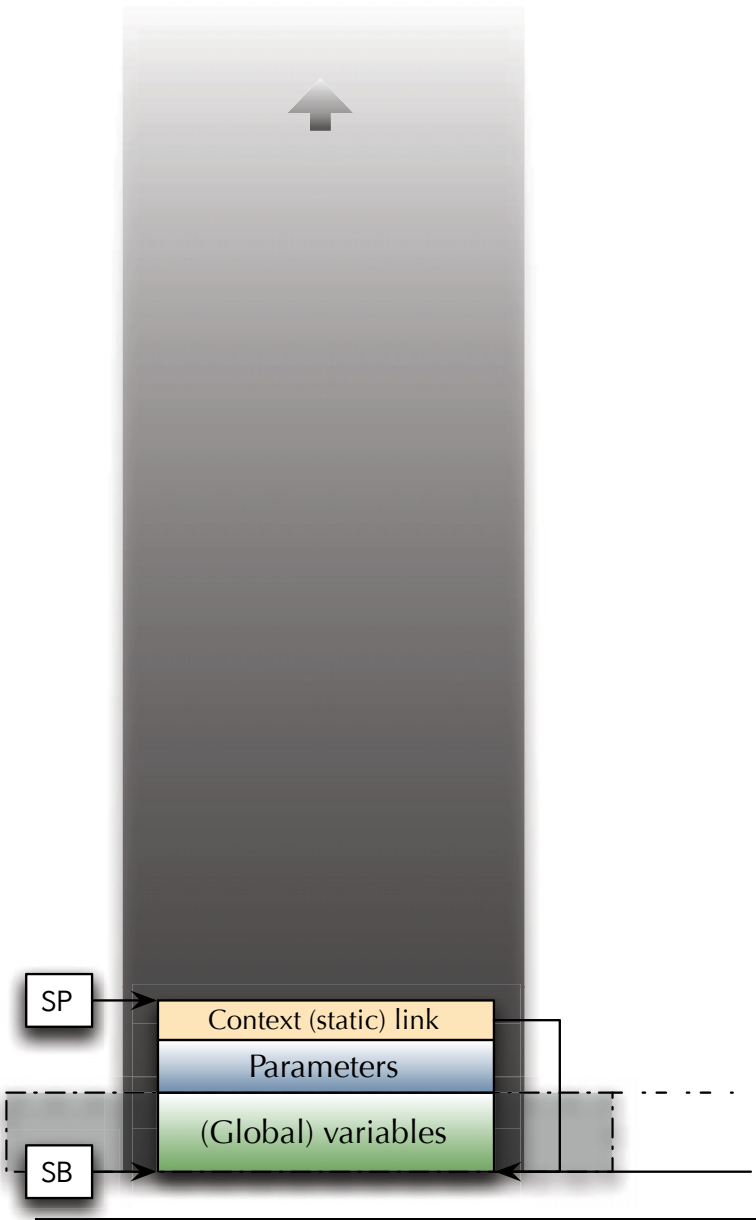


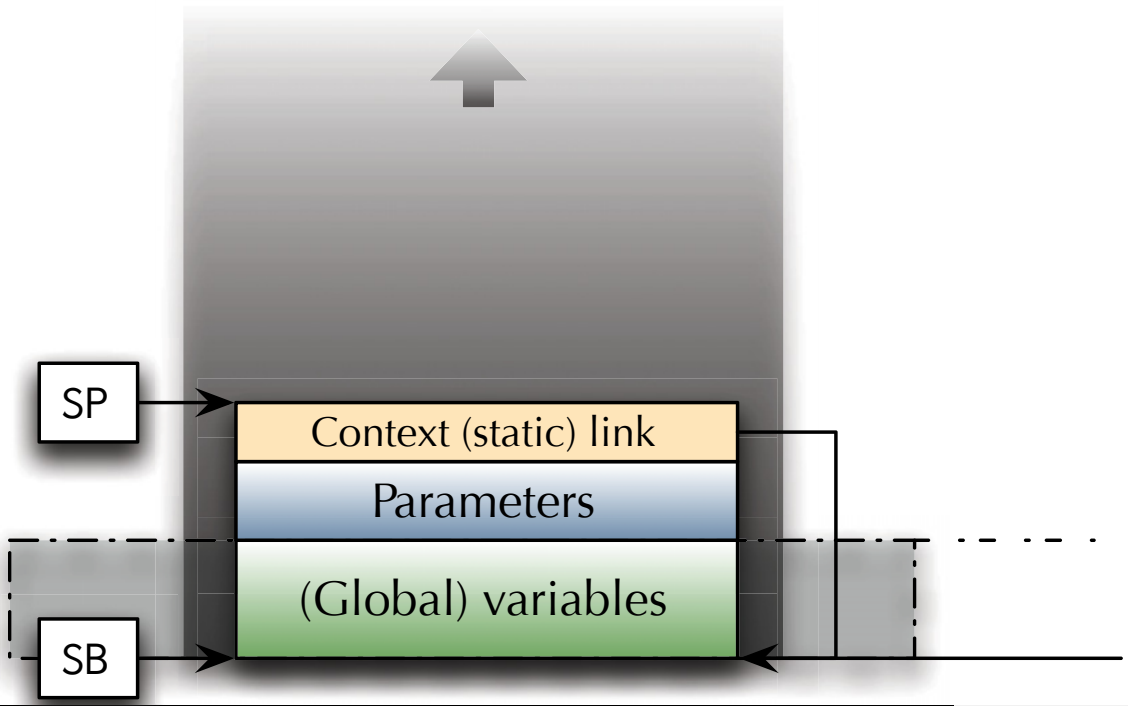
Generic Stack-Frame

Functions (in programming languages) have a context.

☞ E.g. the *surrounding function* or the *hosting object*.

The caller knows this context and provides it.





Generic Stack-Frame

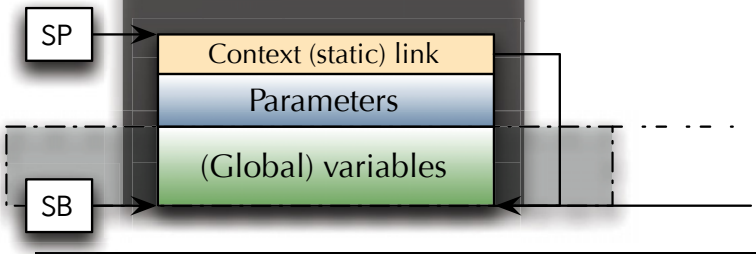
Functions (in programming languages) have a context.

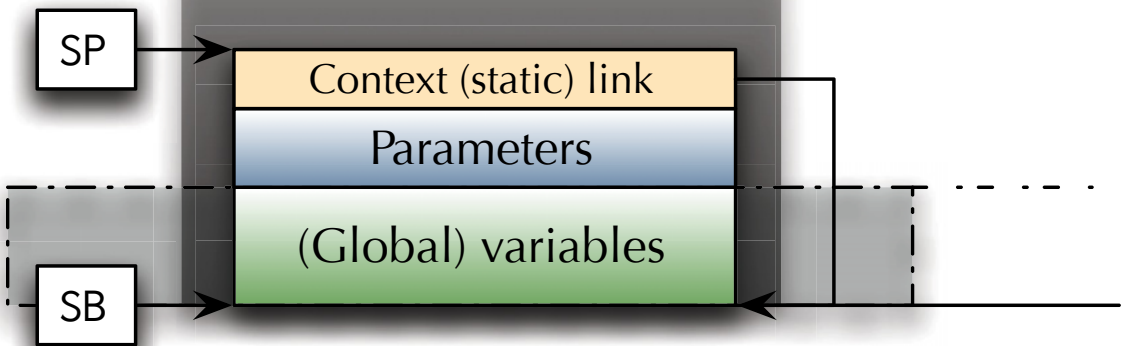
☞ E.g. the *surrounding function* or the *hosting object*.

The caller knows this context and provides it.

☞ Some languages will not have a context by default, like C or Assembly.

(gnu C expands the C standard and provides it though)





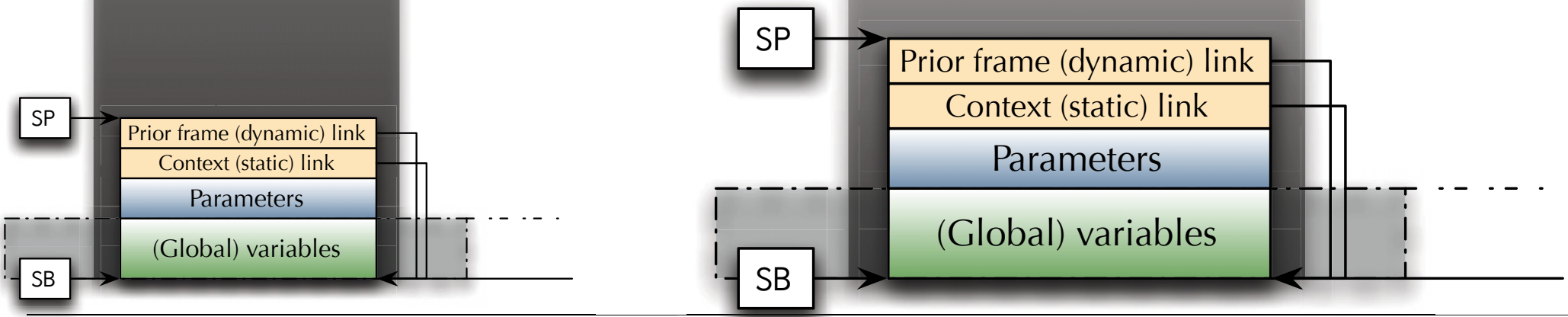
Generic Stack-Frame

The caller also provides a reference to its own stack frame.

☞ This builds a linear chain of calls through the stack.

Will be used e.g. for debugging (stack trace) and exception propagation.

☞ The static and dynamic link might be identical in some cases.



Static vs. dynamic links

```
function a (x : Integer) return Integer is
   function b (y : Integer) return Integer is (x + y);
   function c (z : Integer) return Integer is (b (z));
begin
   return c (x);
end a;
```

```
a :: Integer -> Integer
a x = c x
   where
      b :: Integer -> Integer
      b y = x + y
      c :: Integer -> Integer
      c z = b z
```

Static vs. dynamic links

```
function a (x : Integer) return Integer is
   function b (y : Integer) return Integer is (x + y);
   function c (z : Integer) return Integer is (b (z));
begin
   return c (x);
end a;
```

```
a :: Integer -> Integer
a x = c x
   where
      b :: Integer -> Integer
      b y = x + y
      c :: Integer -> Integer
      c z = b z
```

☞ The **dynamic** and **static link** for `function c` are both `function a`.

Static vs. dynamic links

```
function a (x : Integer) return Integer is
   function b (y : Integer) return Integer is (x + y);
   function c (z : Integer) return Integer is (b (z));
begin
   return c (x);
end a;
```

```
a :: Integer -> Integer
a x = c x
   where
      b :: Integer -> Integer
      b y = x + y
      c :: Integer -> Integer
      c z = b z
```

Dynamic link (prior frame)

☞ The caller of `function` b is `function` c.

☞ Hence exceptions raised in b are handled first in b, then in c, and then a.

☞ The **dynamic** and **static link** for `function` c are both `function` a.

Static vs. dynamic links

```
function a (x : Integer) return Integer is
   function b (y : Integer) return Integer is (x + y);
   function c (z : Integer) return Integer is (b (z));
begin
   return c (x);
end a;
```

```
a :: Integer -> Integer
a x = c x
   where
      b :: Integer -> Integer
      b y = x + y
      c :: Integer -> Integer
      c z = b z
```

☞ The **dynamic** and **static link** for function c are both function a.

Dynamic link (prior frame)

☞ The caller of function b is function c.

☞ Hence exceptions raised in b are handled first in b, then in c, and then a.

Static link (context)

☞ The context for function b is function a.

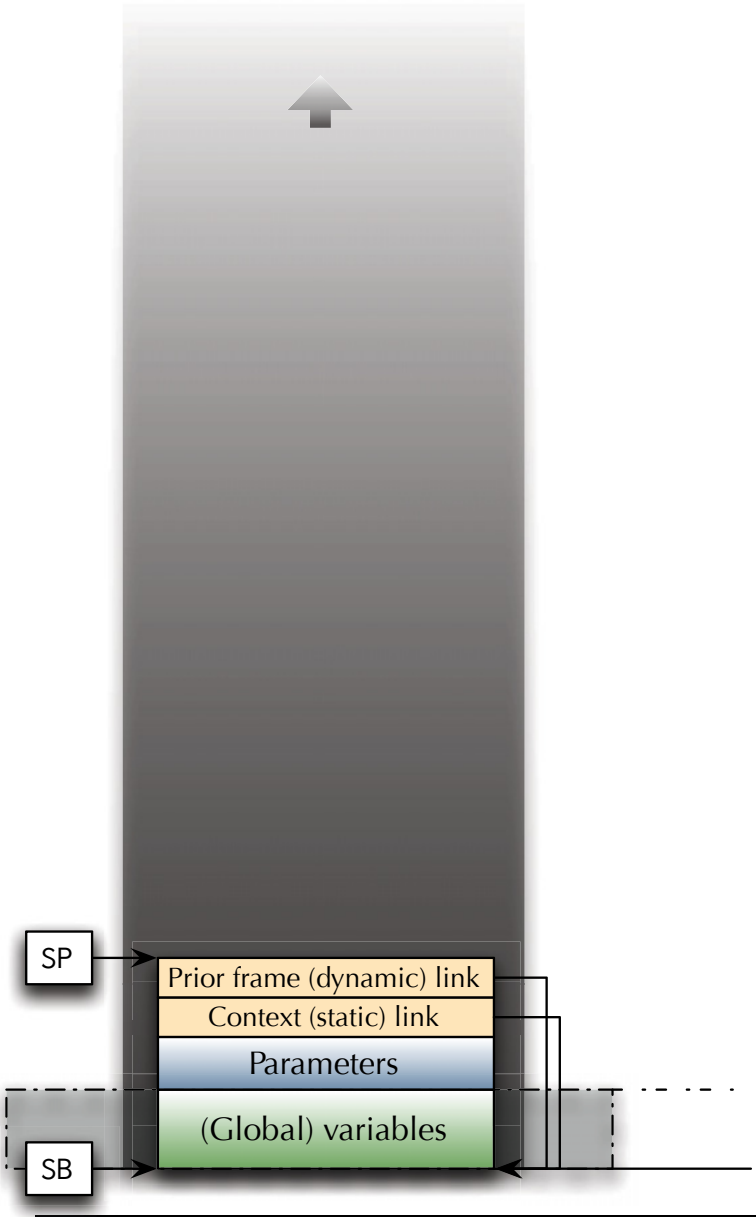
☞ Hence b can access x (but not z).

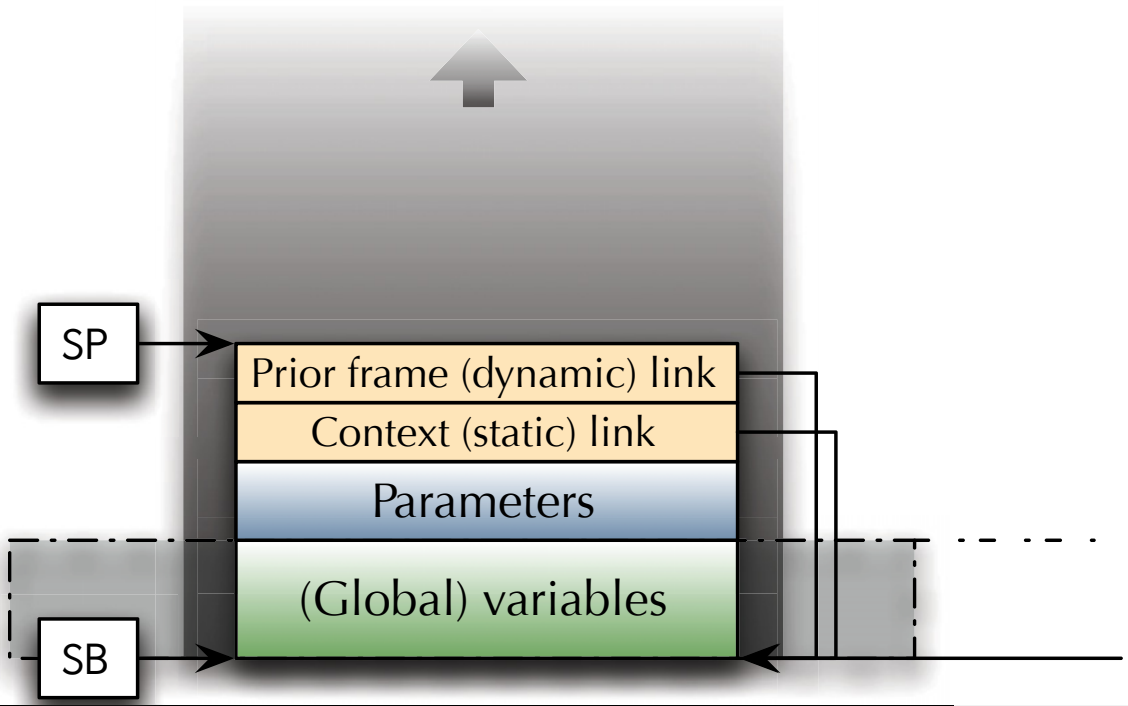
Generic Stack-Frame

The caller also provides a reference to its own stack frame.

☞ This builds a linear chain of calls through the stack.

Will be used e.g. for debugging (stack trace) and exception propagation.

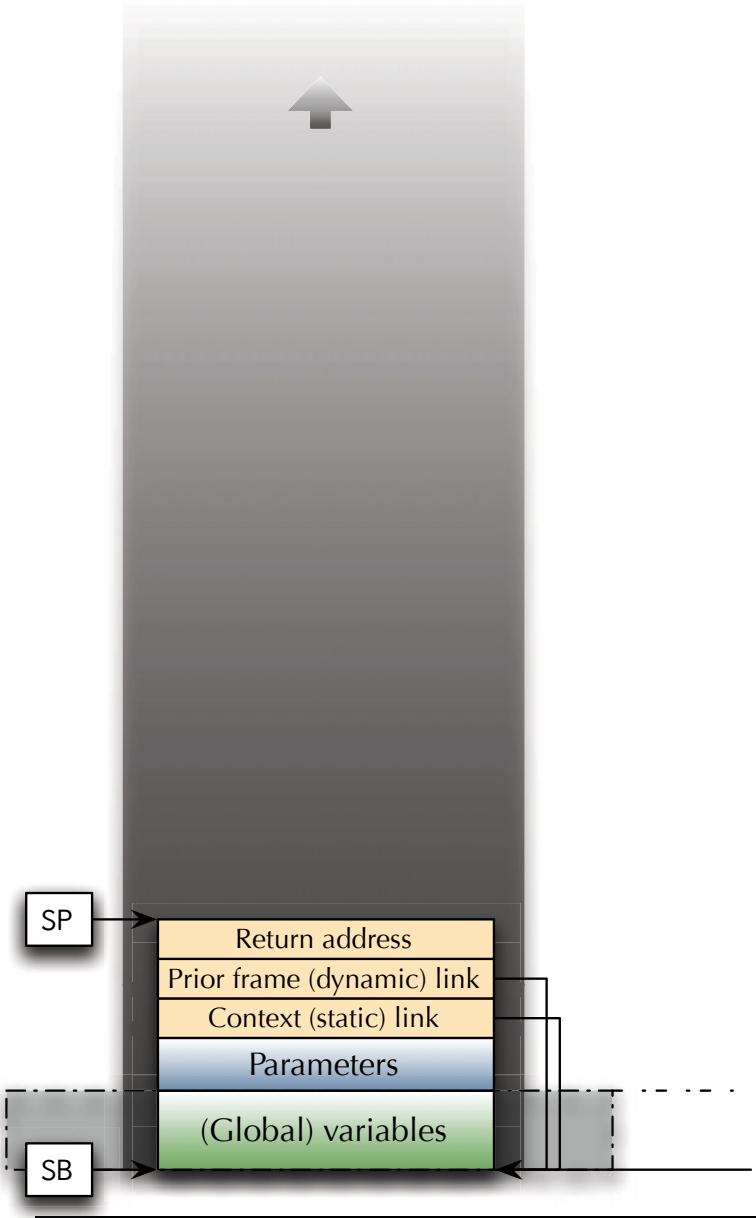


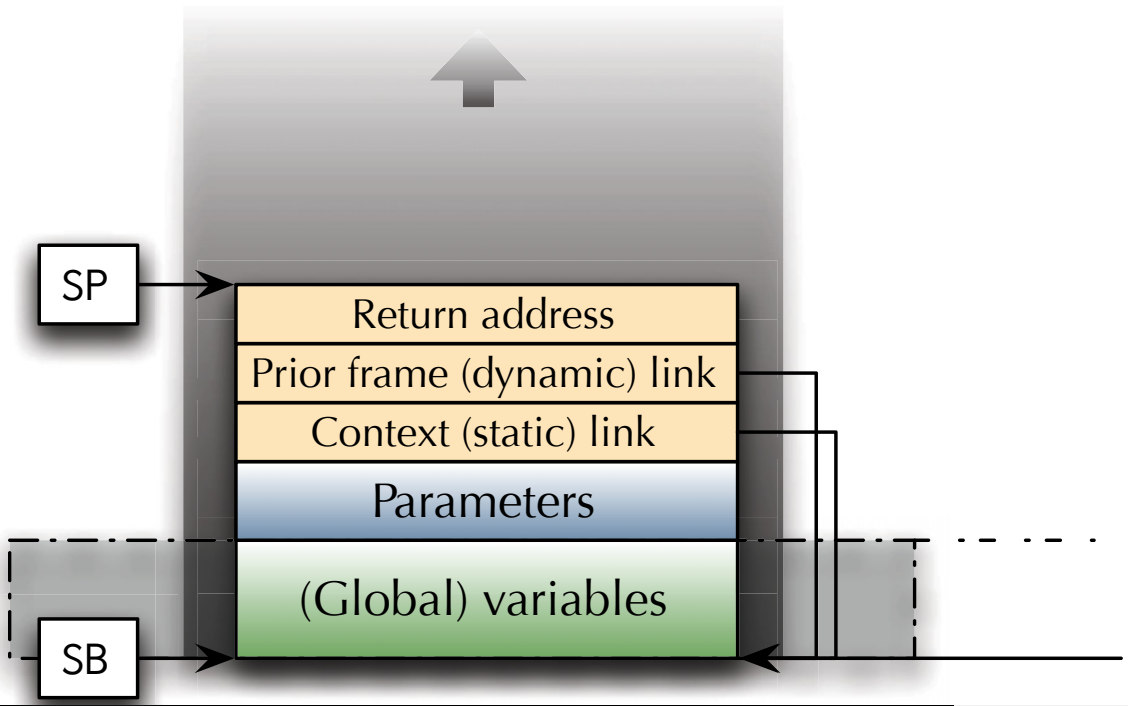


Generic Stack-Frame

The last item to be stored before handing over control is the address of the following instruction.

☞ The control flow can later return to this address.



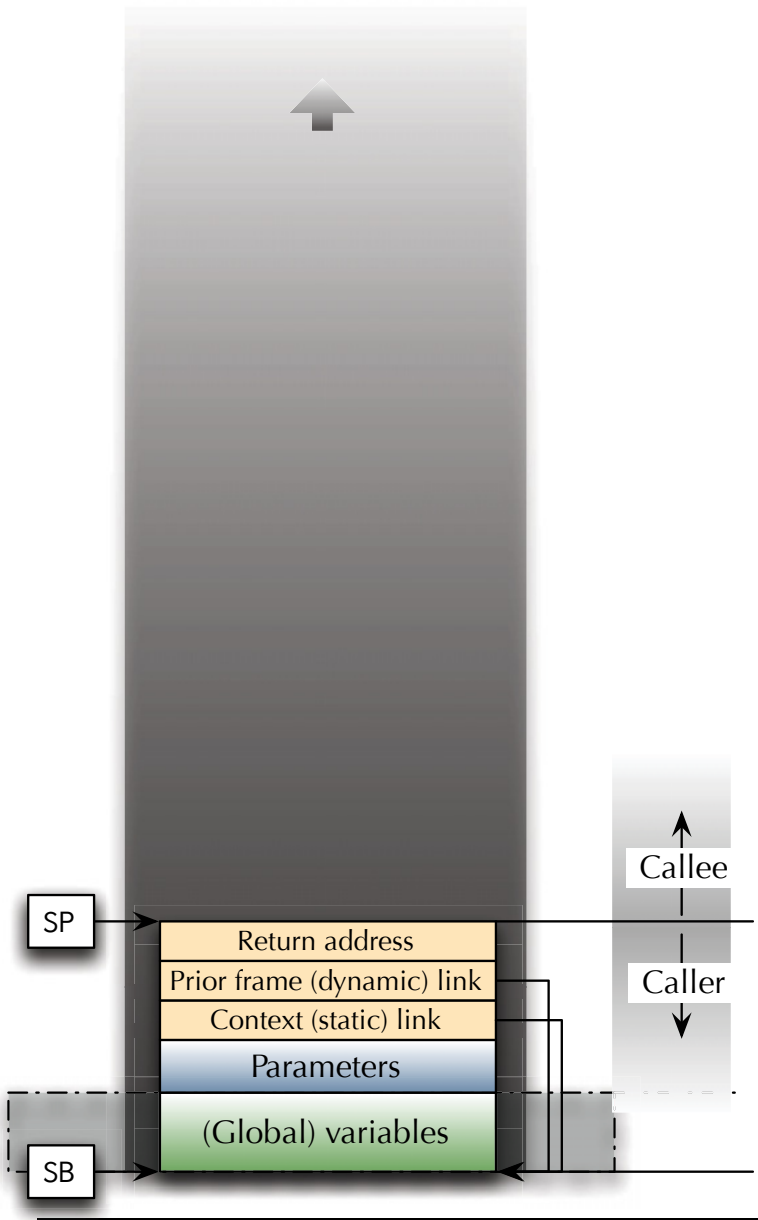


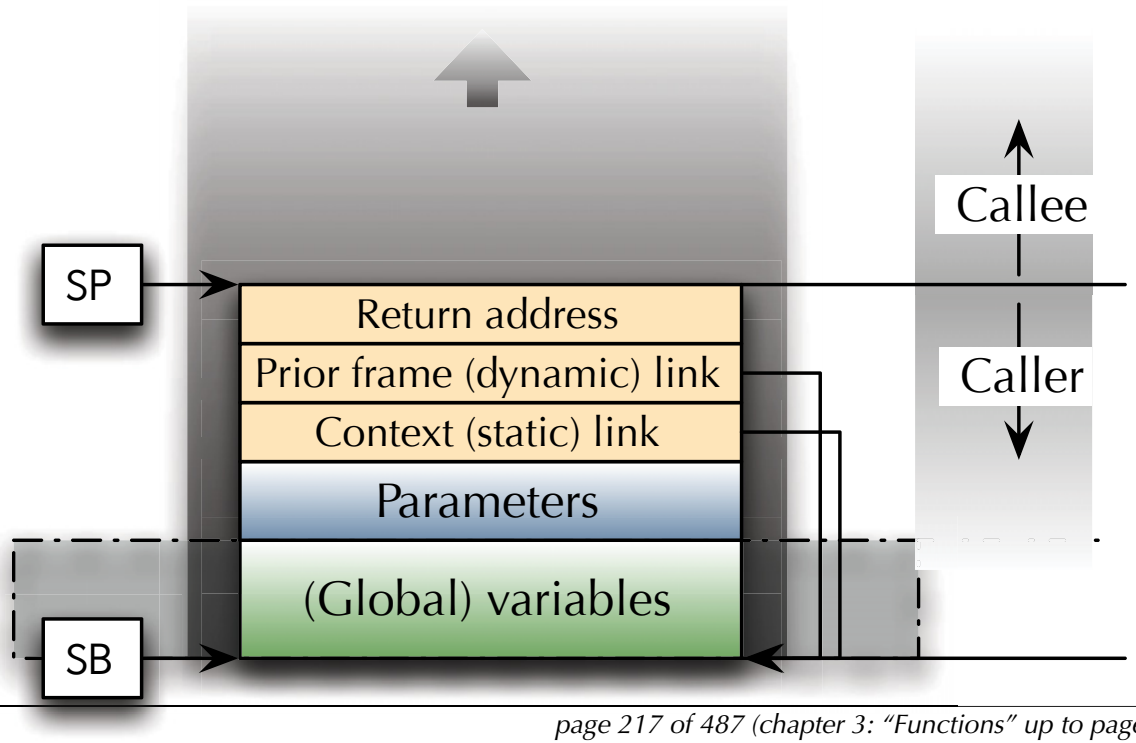
Generic Stack-Frame

Control is handed over to the callee.

☞ The Instruction Pointer (IP), sometimes also called Program Counter (PC) is changed to a new address.

Operations from here are in the control of the callee.



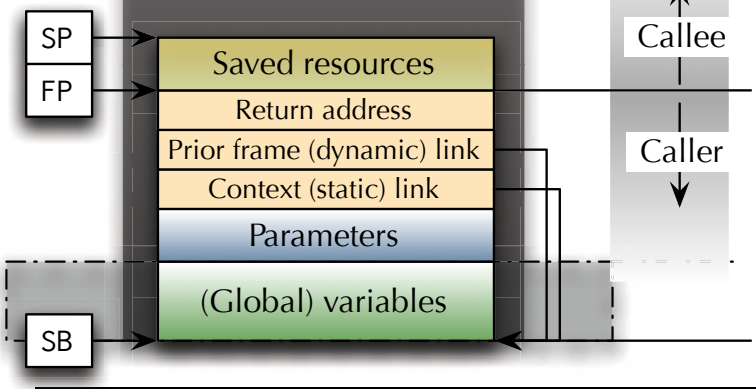


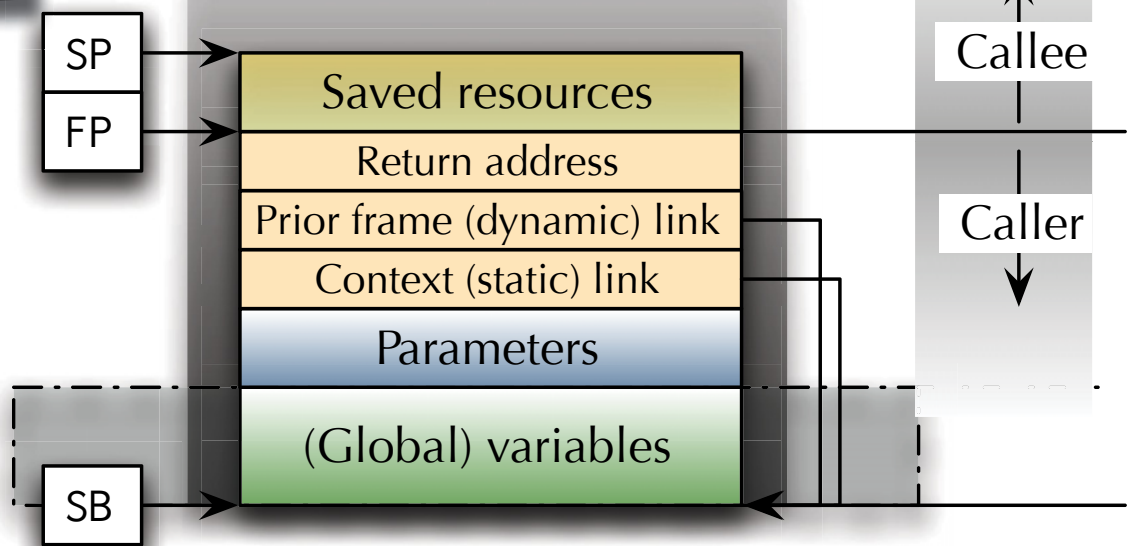
Generic Stack-Frame

A **Frame Pointer** (FP) is established at the boundary between the caller and the callee.

- ☞ Upwards from the FP: data from this function.
- ☞ Downwards from the FP: data provided by the previous function.

Saved resources are for instance registers which the callee is planning to use.



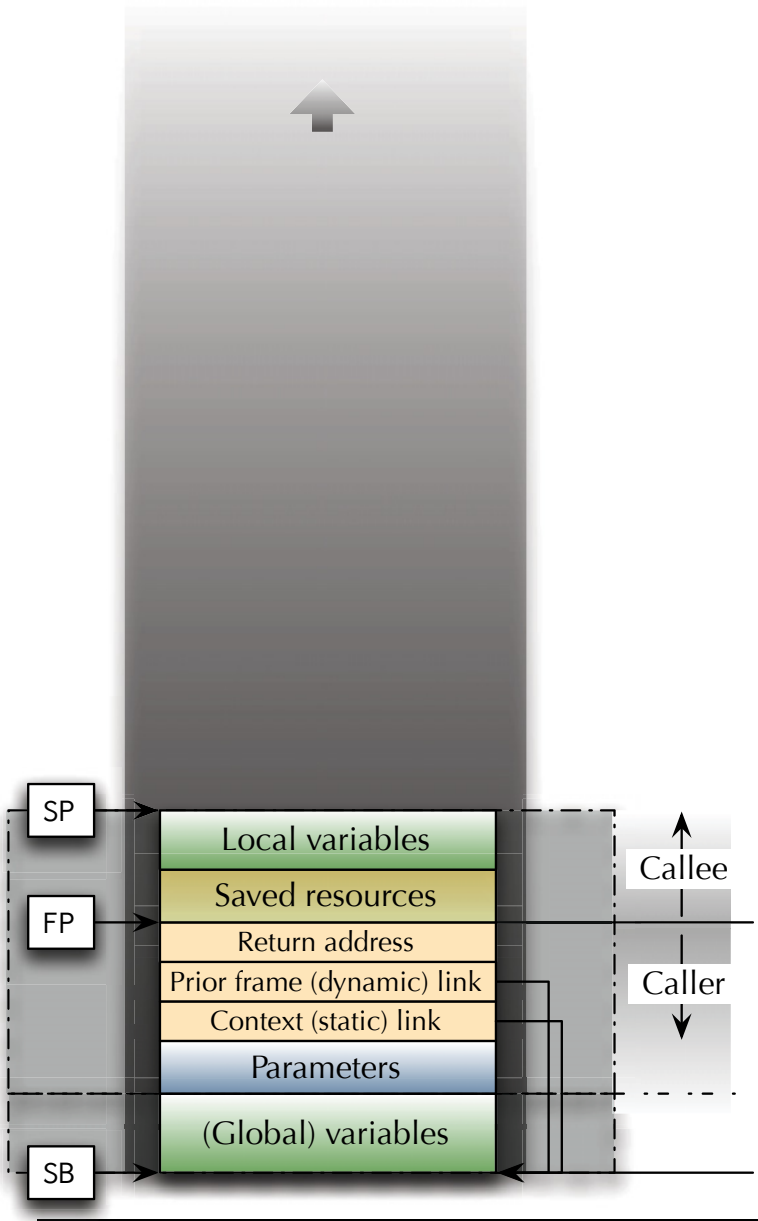


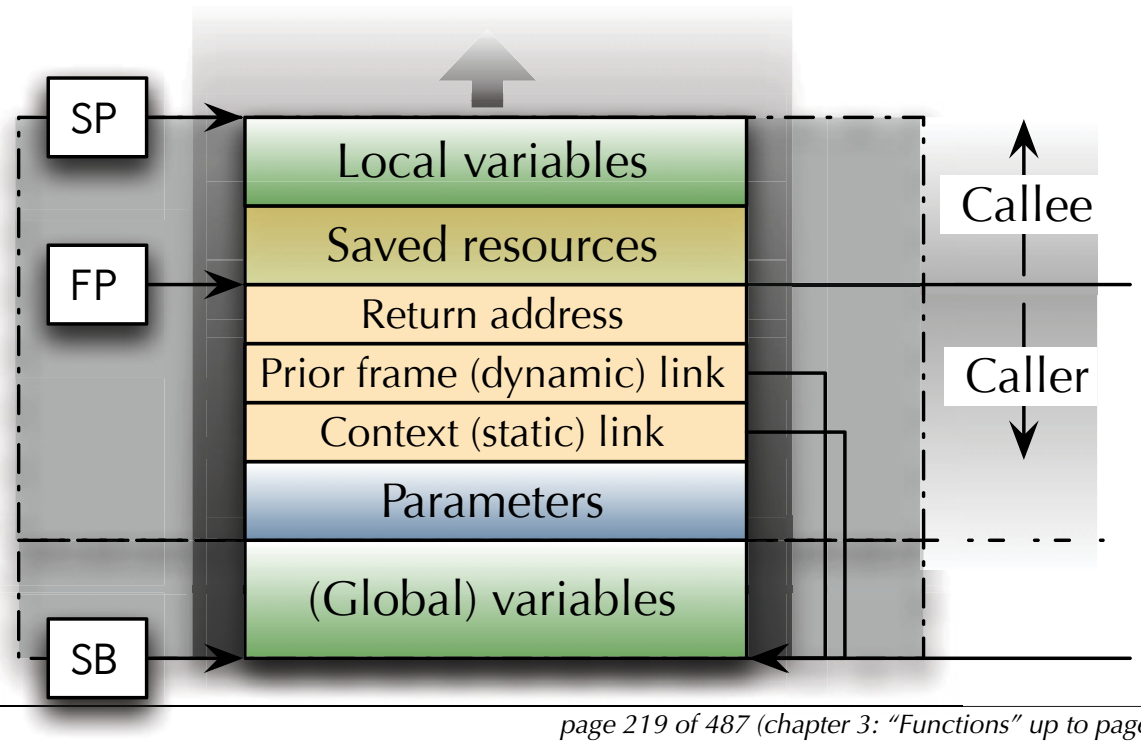
Generic Stack-Frame

Local variables are allocated (by moving the stack pointer).

Local variables can be of any size or structure, unless the stack overflows.

☞ The completes a new **stack frame**.





Generic Stack-Frame

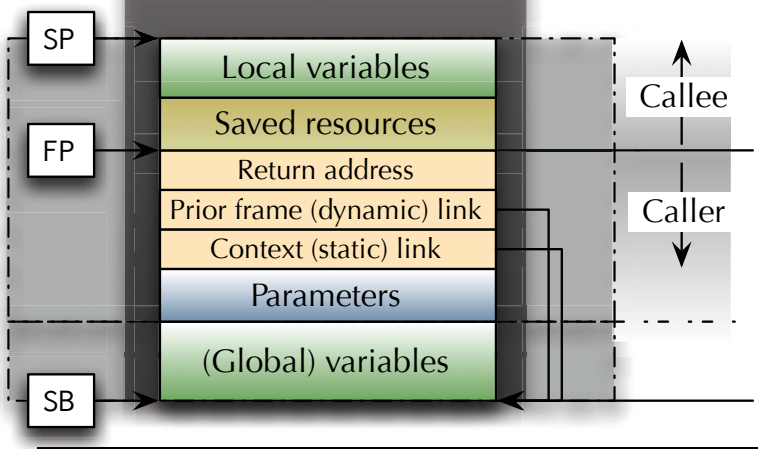
Local variables are allocated (by moving the stack pointer).

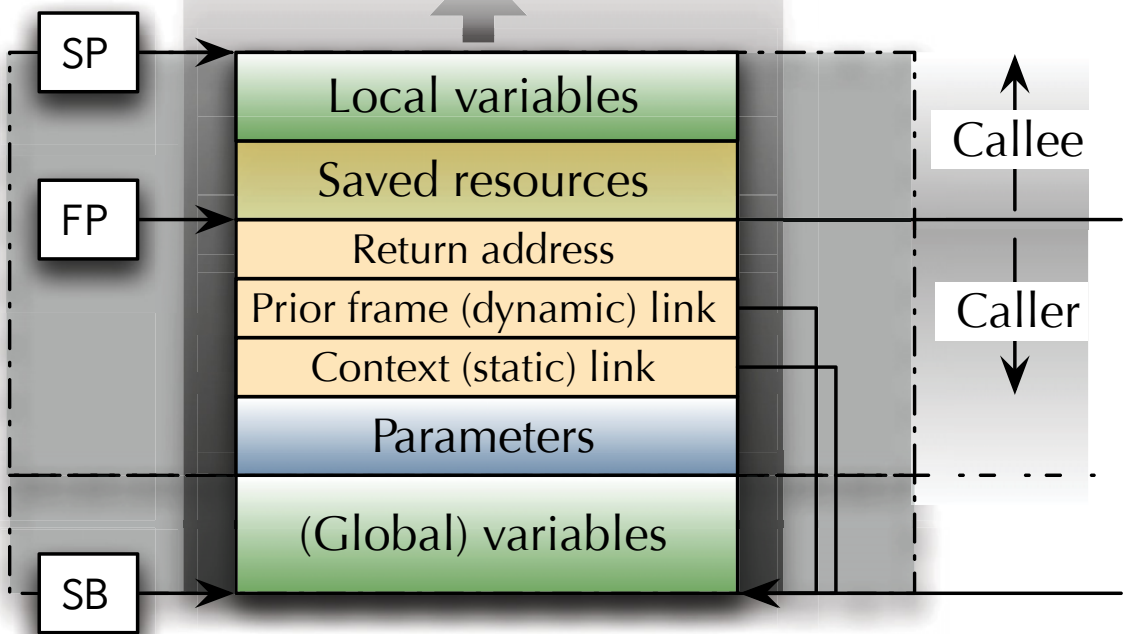
Local variables can be of any size or structure, unless the stack overflows.

☞ The completes a new **stack frame**.

While this function is executing, local variables can still be added.

☞ Handy, if e.g. the size of a local variable is not yet determined when the function starts.

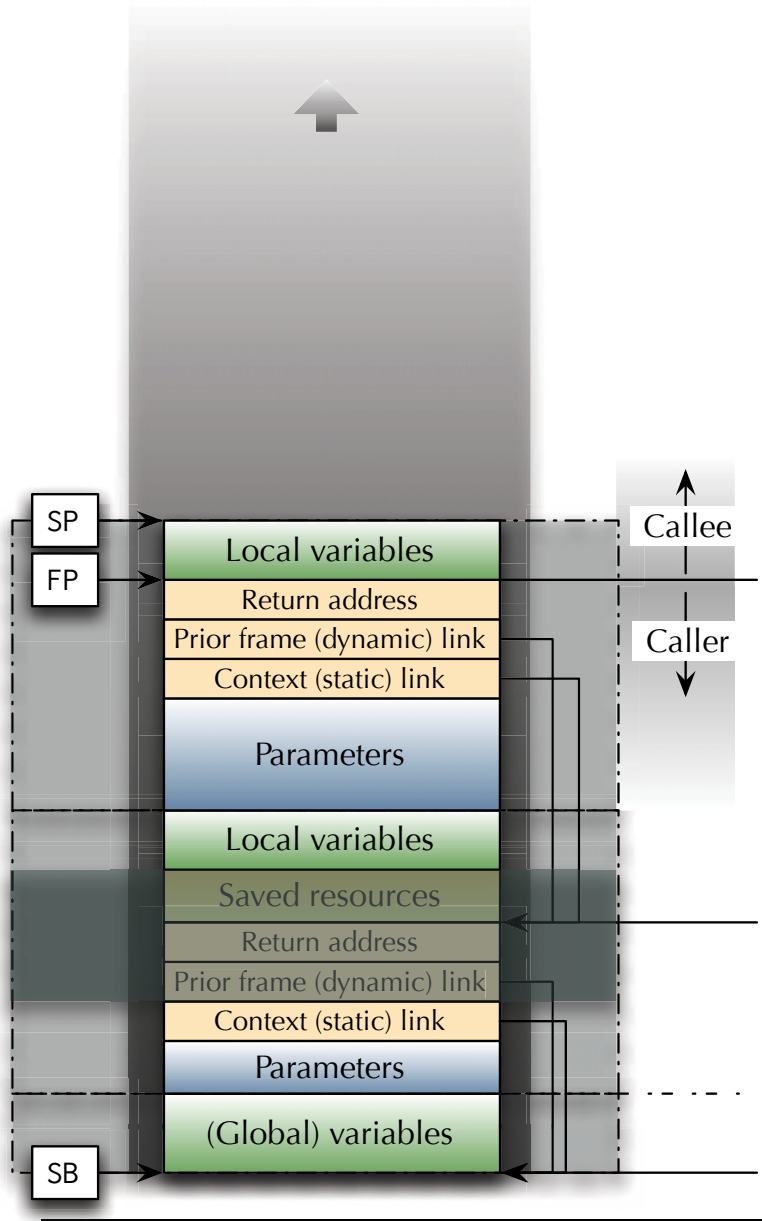


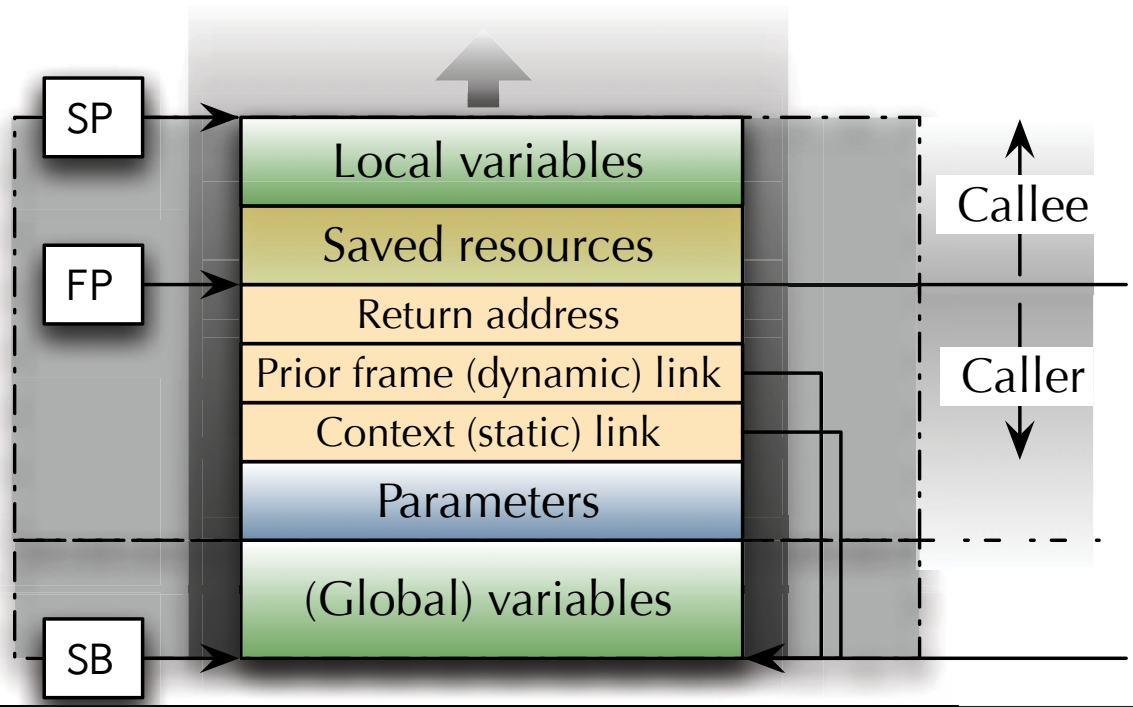


Generic Stack-Frame

The next function call will produce the next stack frame.

☞ Variables and parameters from the context stay visible (via the chain of static links).



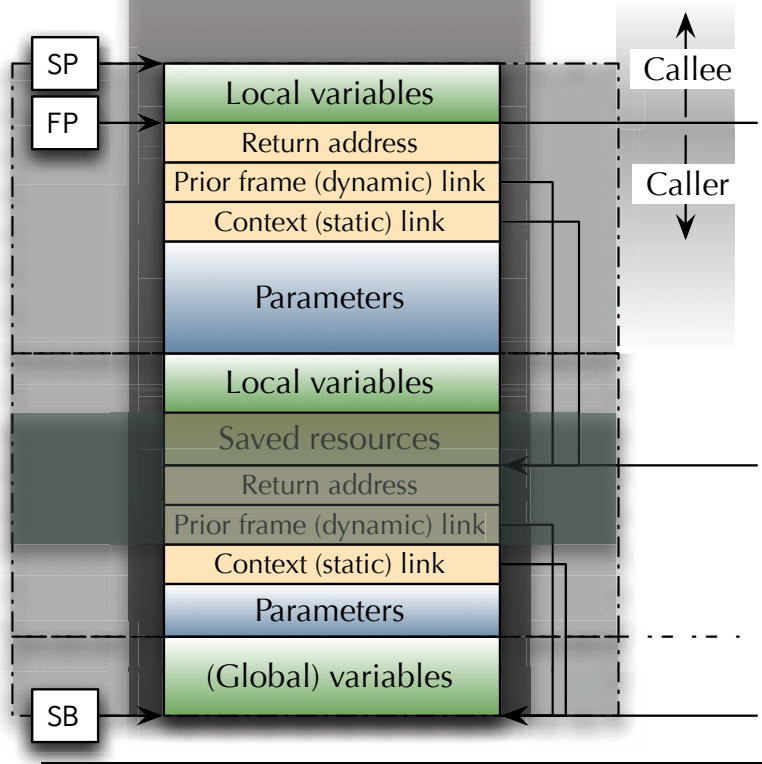


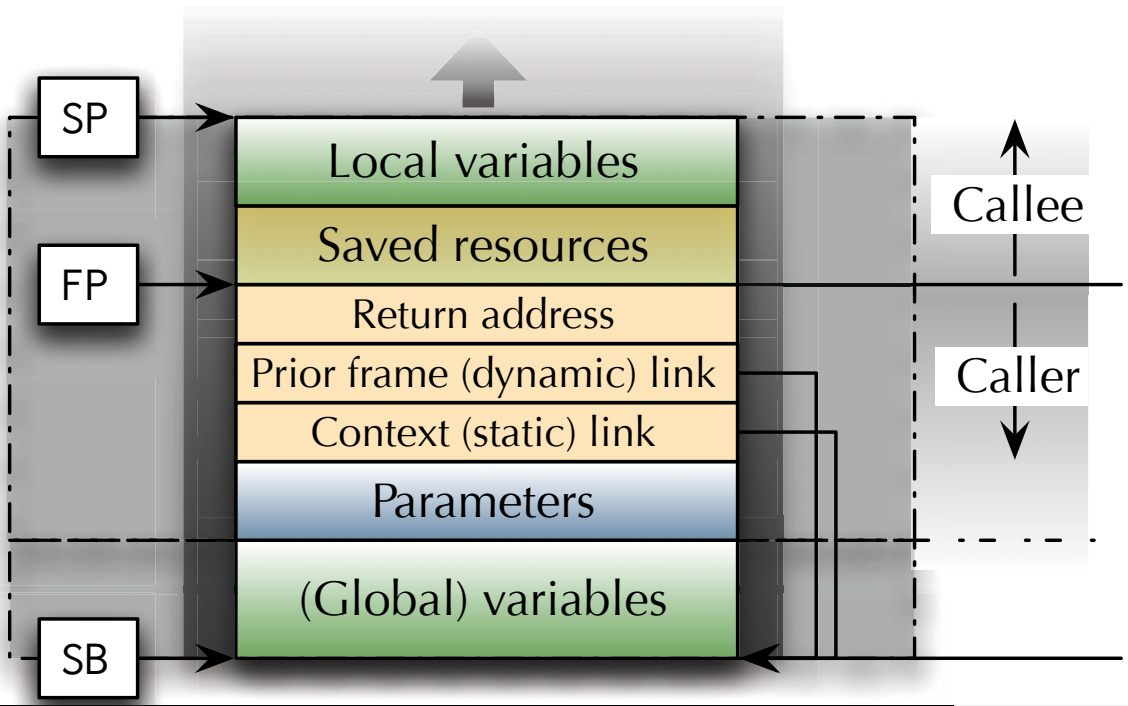
Generic Stack-Frame

The next function call will produce the next stack frame.

☞ Variables and parameters from the context stay visible (via the chain of static links).

Local variables can only be added to the currently executing function.

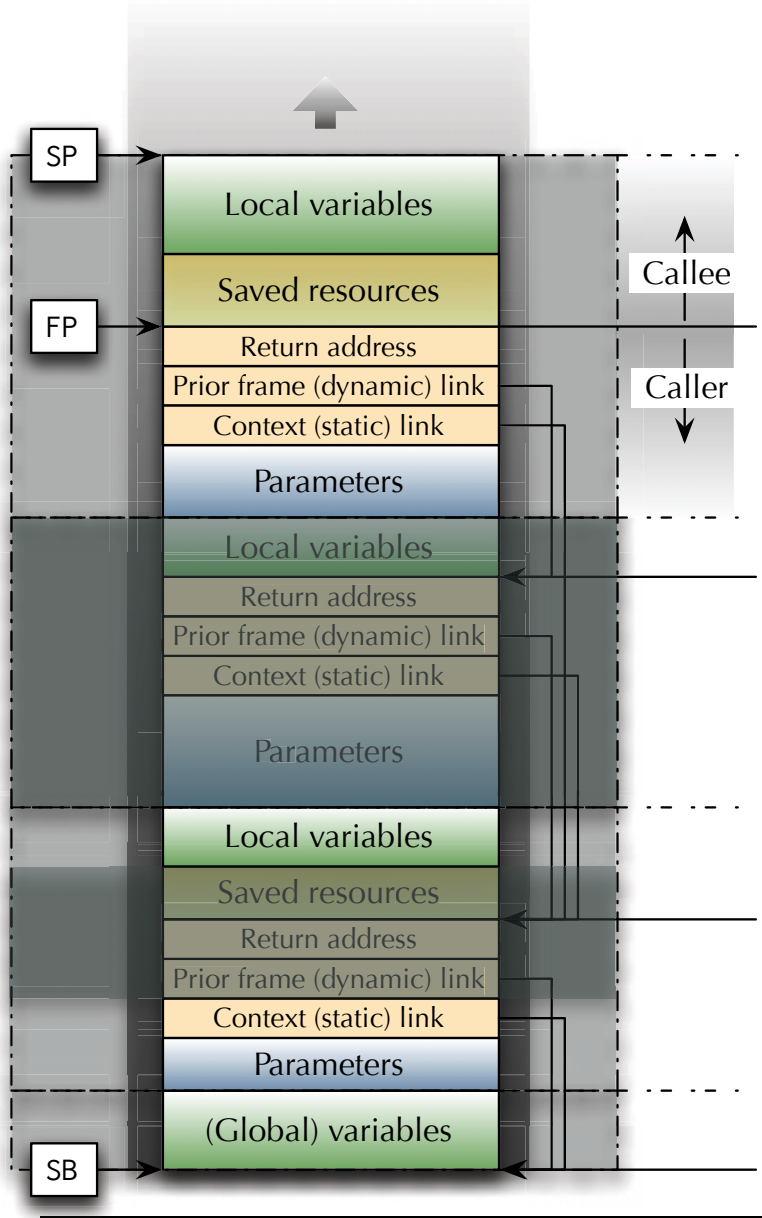


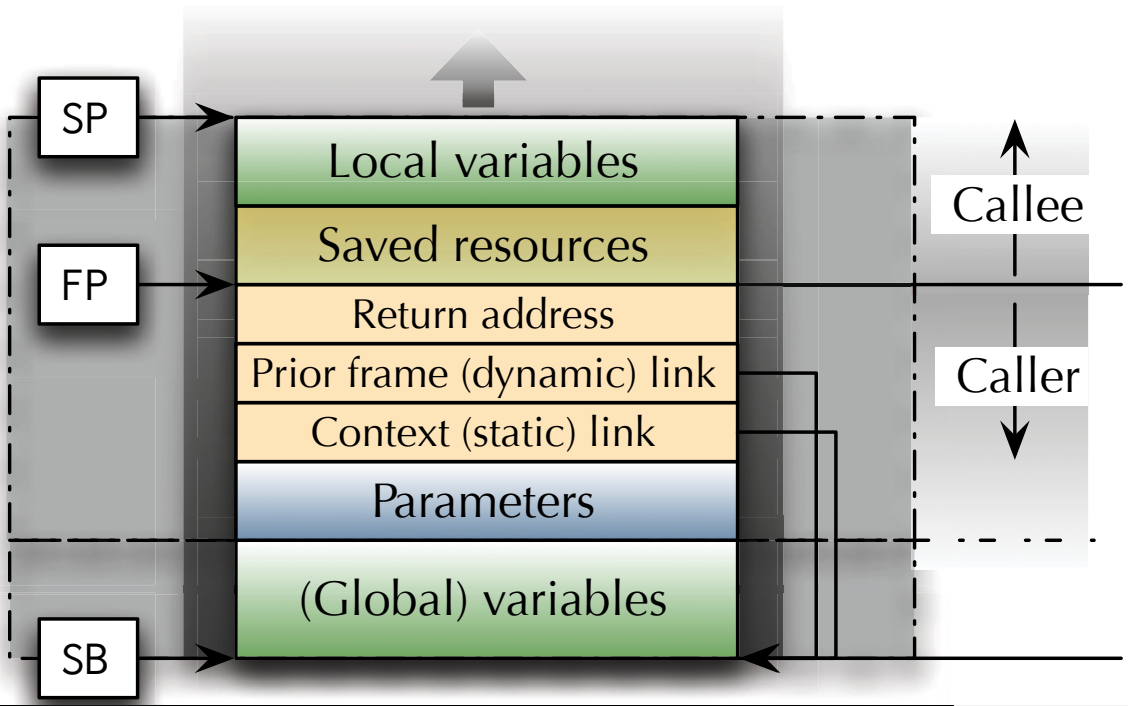


Generic Stack-Frame

The next function call will produce the next stack frame.

☞ Note which variables and parameters are visible.

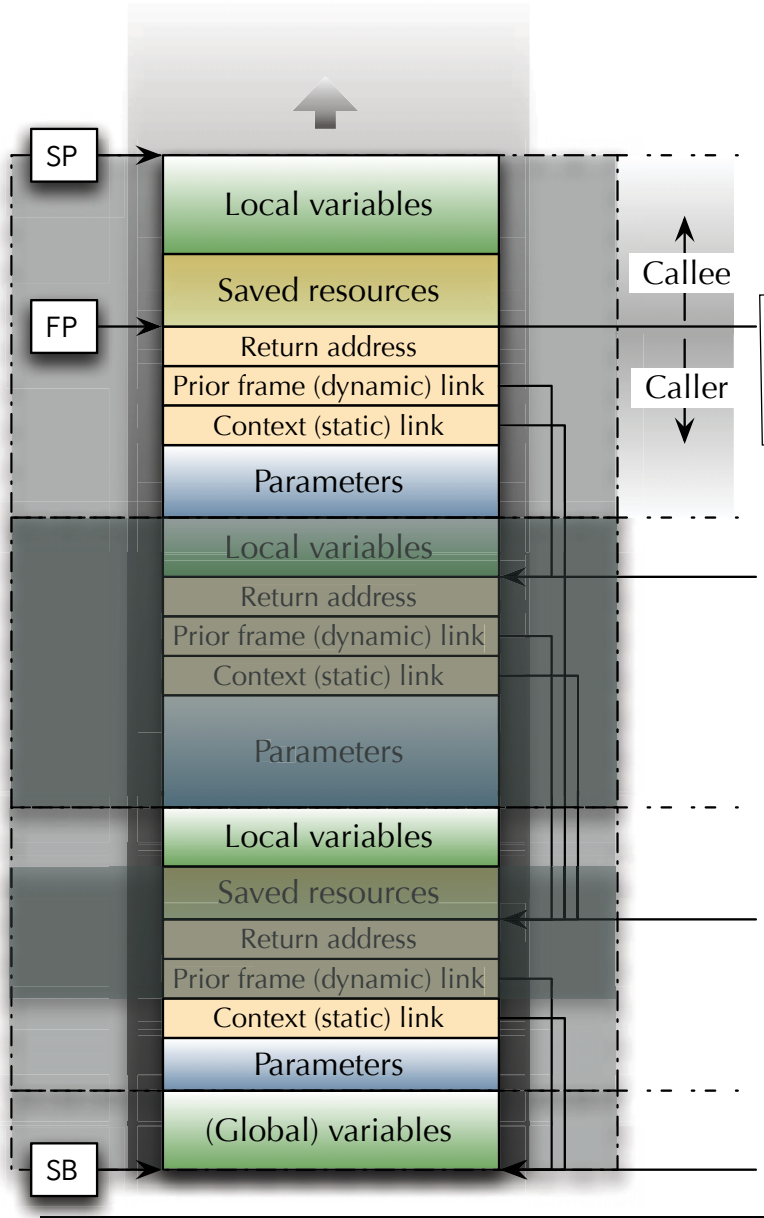




Generic Stack-Frame

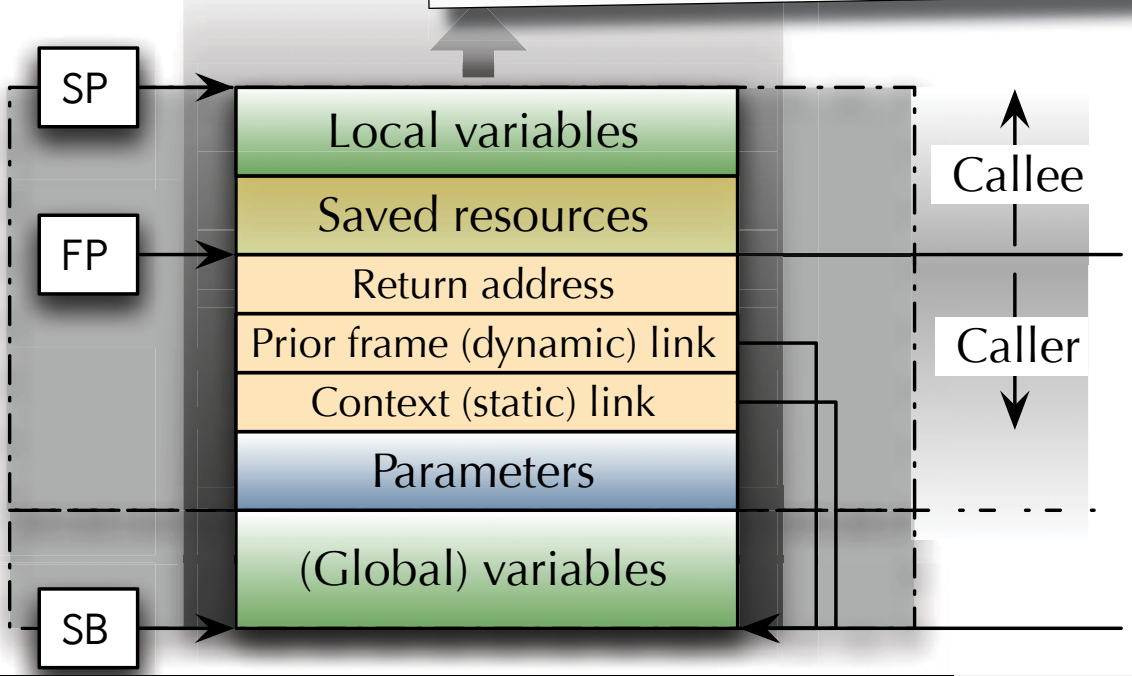
The next function call will produce the next stack frame.

☞ Note which variables and parameters are visible.

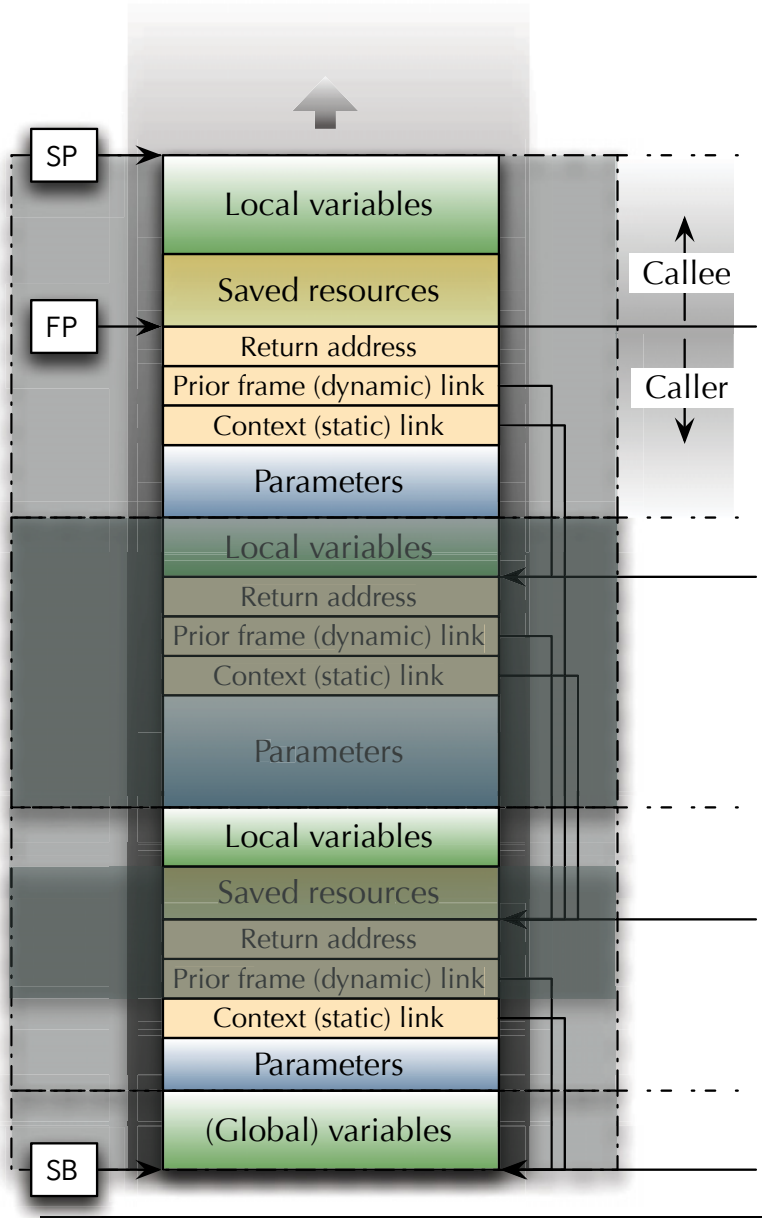


Accessing the context like that can be inefficient!

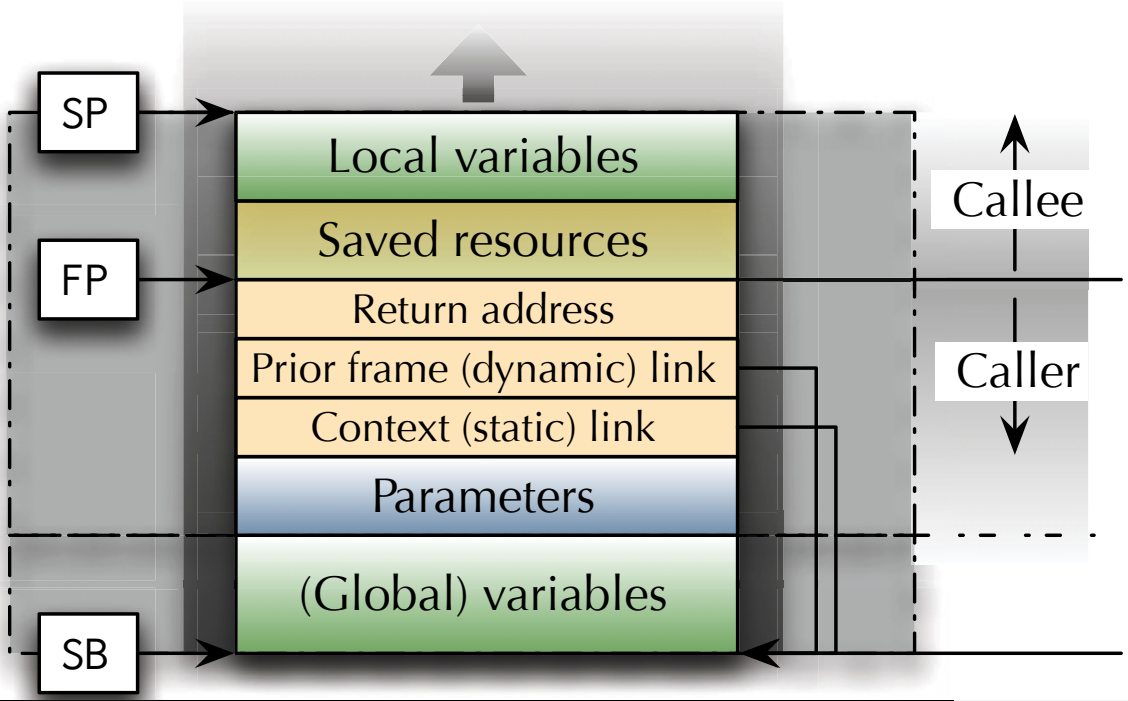
☞ Compilers may chose other mechanism (e.g. **displays**, which make all context levels accessible at once).



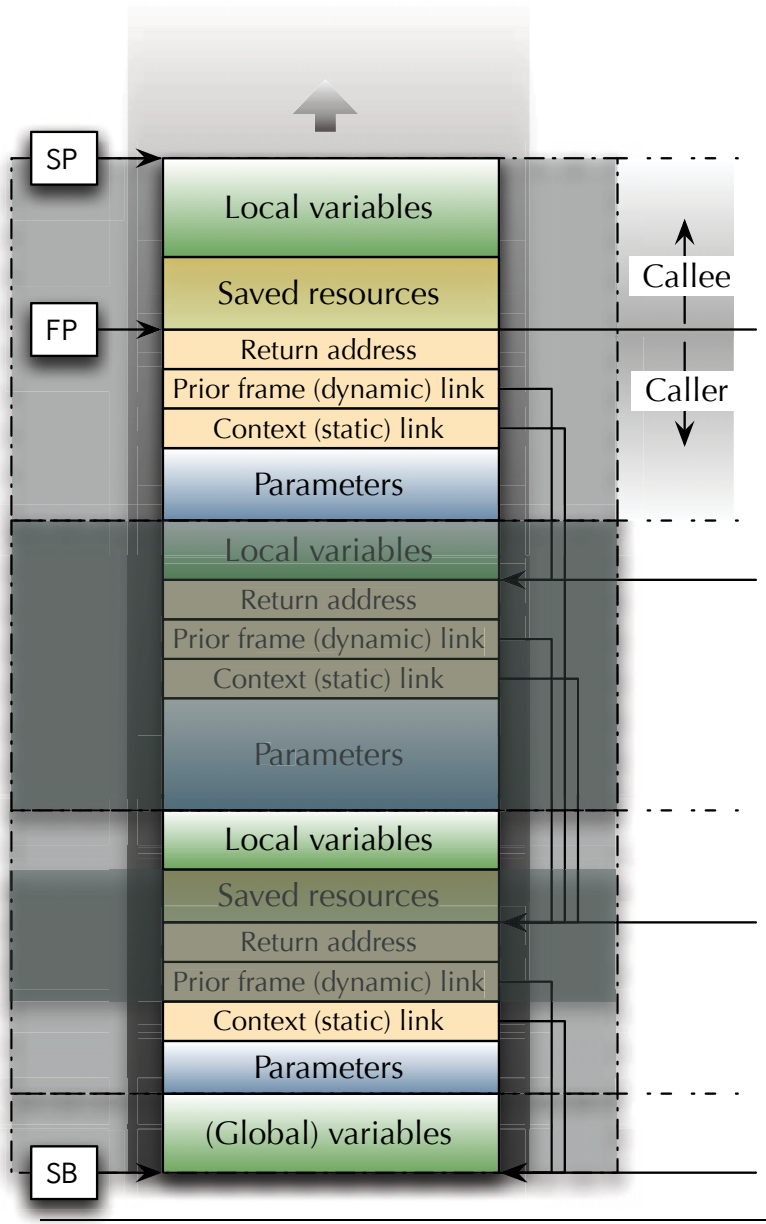
Generic Stack-Frame



☞ How fast / complex is the **allocation** and **deallocation** of *local variables* and *parameters* on the stack?



Generic Stack-Frame – Caller



Pre_Call:

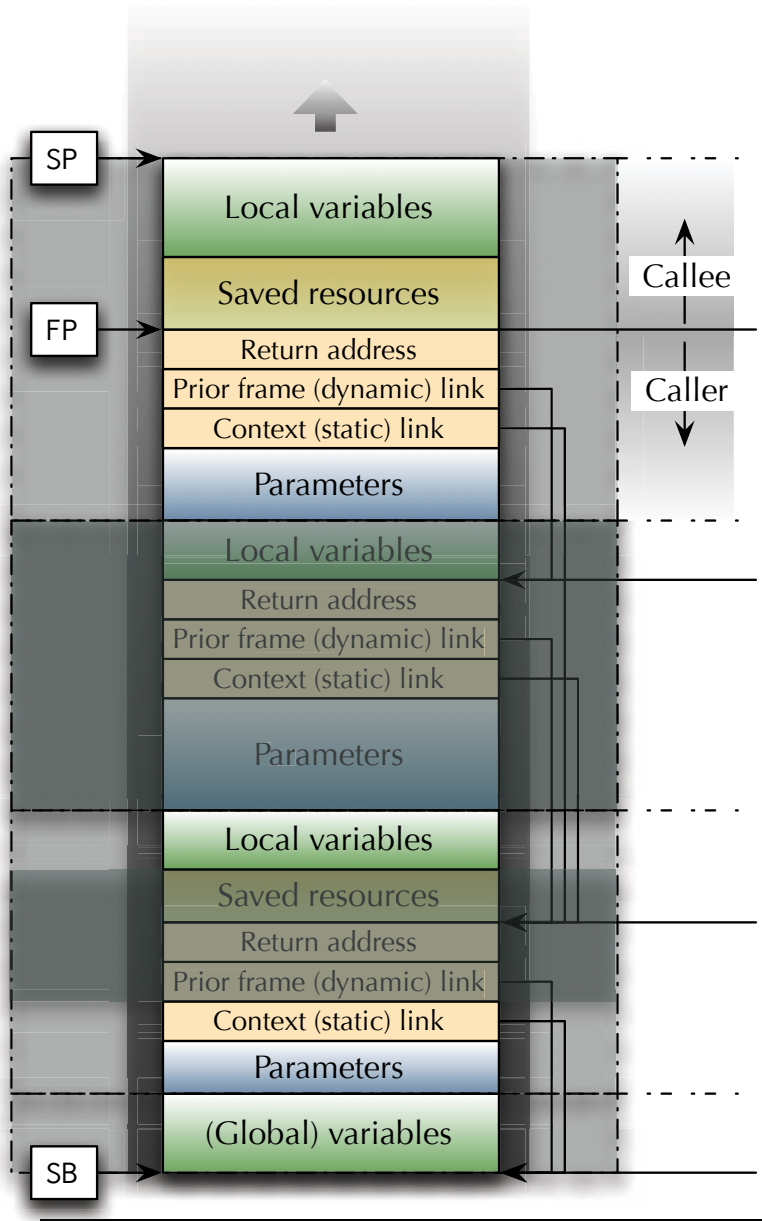
```
…      ; Allocate/identify space for the parameters
…      ; Copy the in and in-out
…      ;    parameters to this space
…      ; Potentially provide links
…      ; Provide a return address ("Post_Call")
…      ;    (usually implicit with the call itself)
```

☞ Call the function

Post_Call:

```
…      ; Copy the out and in-out parameters
…      ;    to local variables or registers
…      ; Potentially restore the frame pointer
…      ; Restore the stack to its previous state
…      ;    (if the stack has been used)
```

Generic Stack-Frame – Callee



Prologue:

```
…      ; Save all registers which are needed
…      ;    inside this function to the stack
…      ; Establish a new frame pointer
…      ;    while potentially saving the previous fp
…      ; Allocate/identify space for local variables
…      ;    Potentially initialize local variables
```

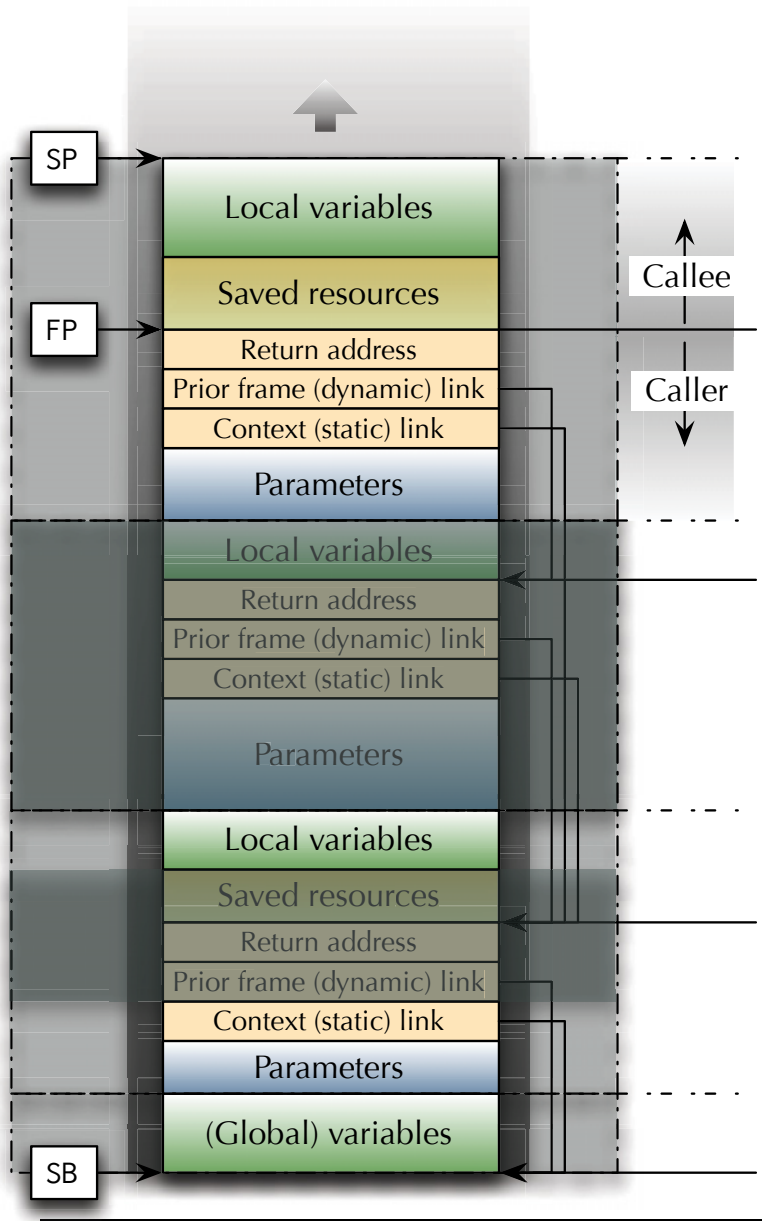
☞ Operations, which will use
local and context variables and parameters
(via the FP(s))

Epilogue:

```
…      ; Potentially restore the prior frame pointer
…      ; Restore the stack to its state at entry
```

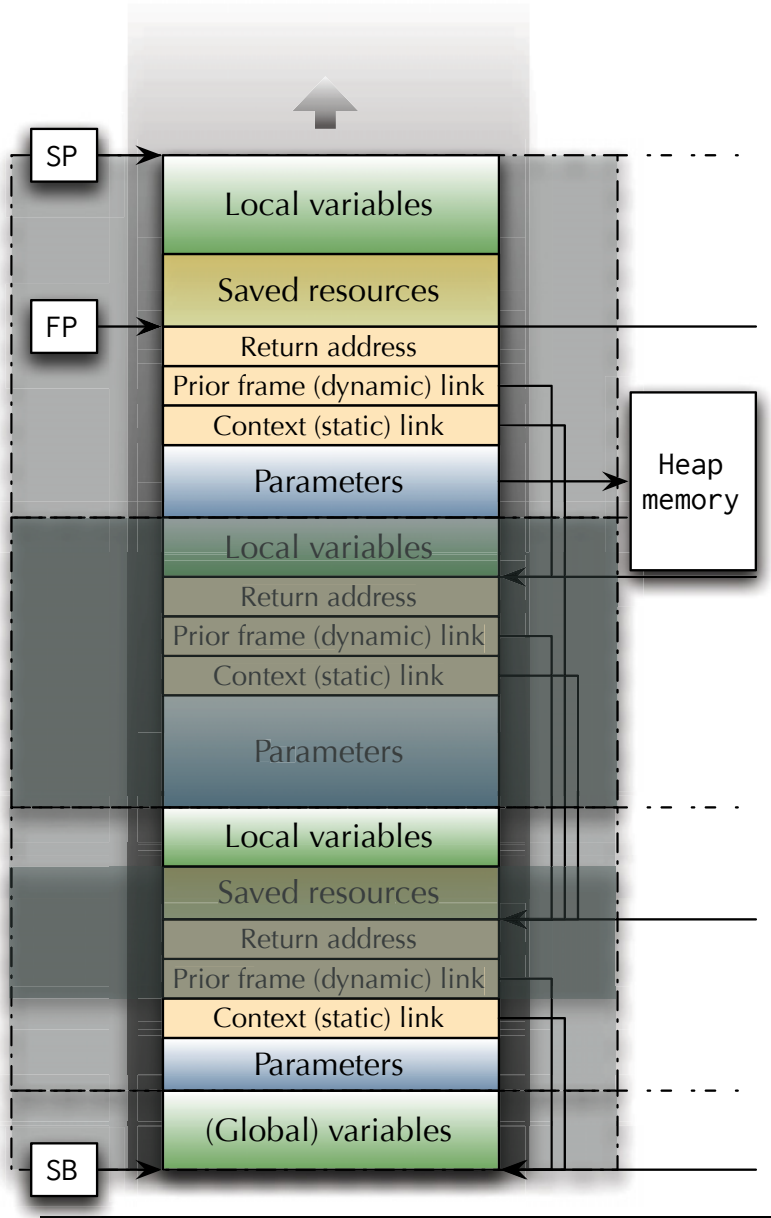
☞ Return from function

Generic Stack-Frame – Heap



☞ How to keep any memory allocation after function return?

Generic Stack-Frame – Heap

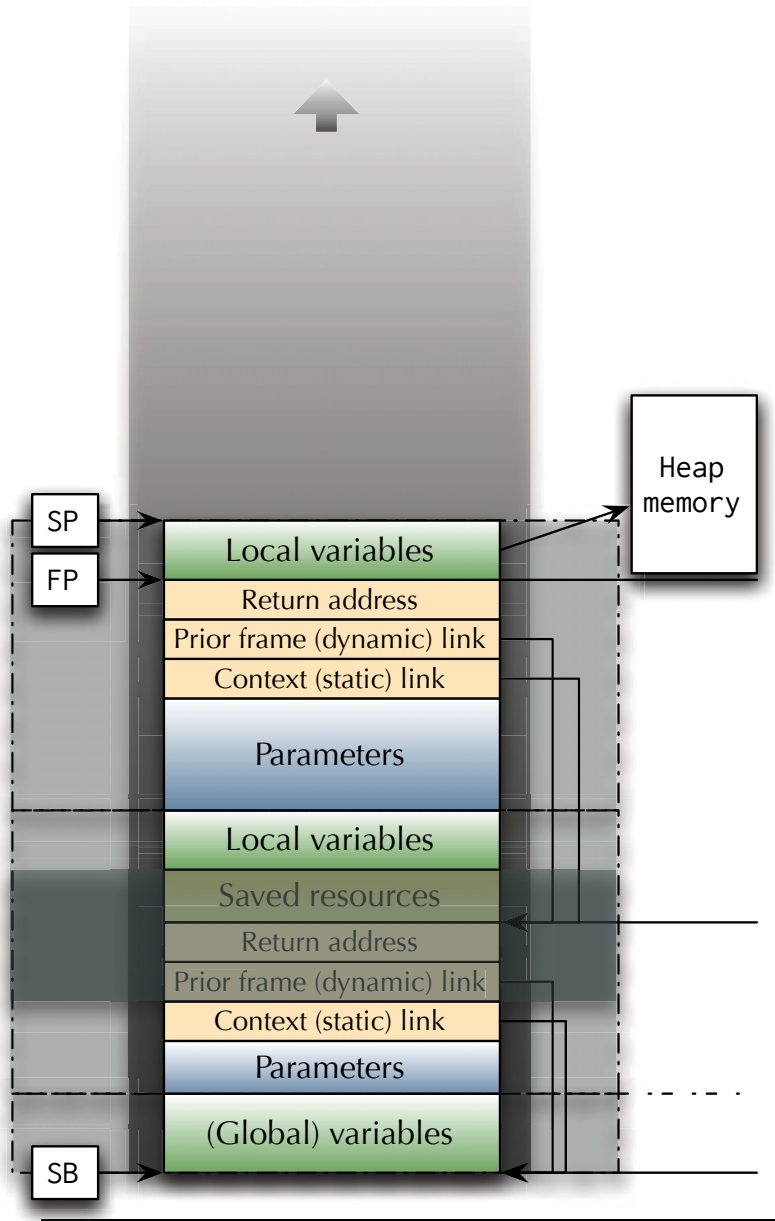


☞ How to keep any memory allocation after function return?

☞ By using an out, by-reference parameter, the link to the newly allocated memory area is kept.

Generic Stack-Frame – Heap

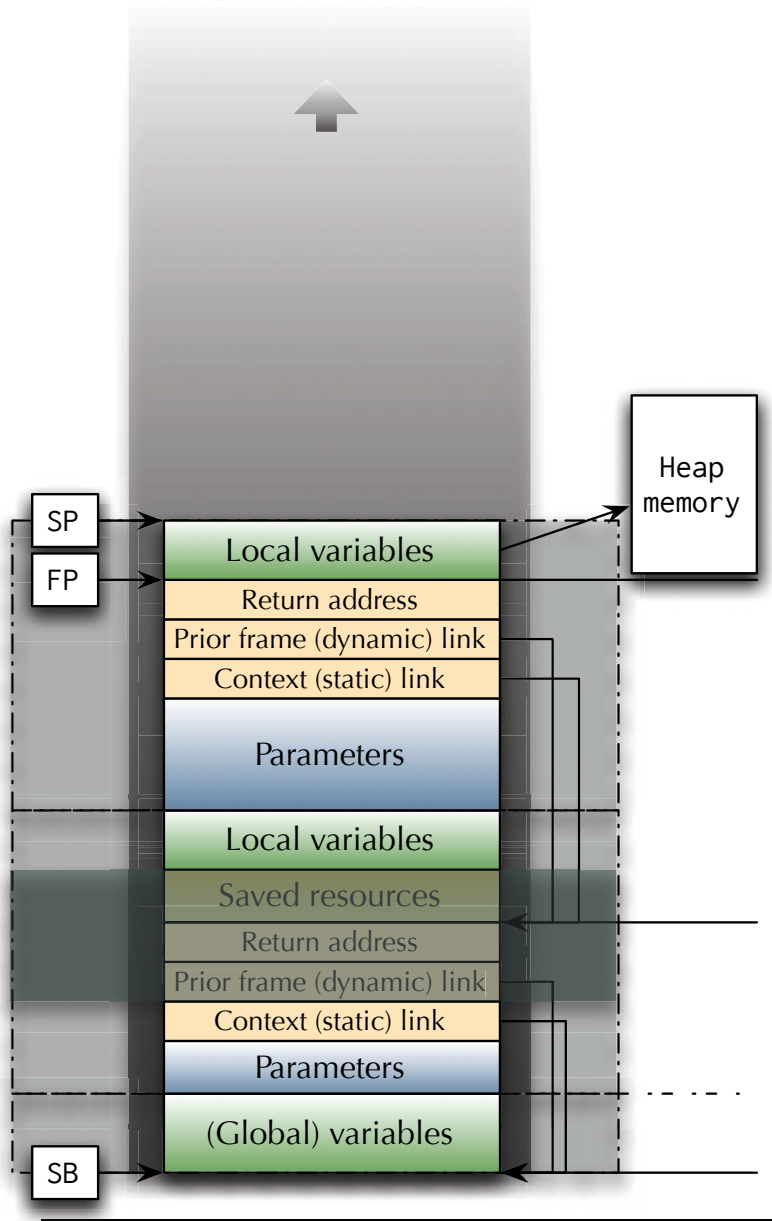
☞ How to keep any memory allocation after function return?



☞ By using an out, by-reference parameter, the link to the newly allocated memory area is kept.

☞ … and a local variable in the calling function can keep this link.

Generic Stack-Frame – Heap



☞ How to keep any memory allocation after function return?

☞ By using an out, by-reference parameter, the link to the newly allocated memory area is kept.

☞ … and a local variable in the calling function can keep this link.

☞ When to deallocate though?

- Garbage collection (Java)?
- Smart pointers (C++)?
- Reference ownerships (Rust)?
- Scoped pointers / storage pools (Ada)?

Summary

Functions

- **Framework**
  - Return address
  - Relative addressing
- **Parameter passing modes and mechanisms**
  - Copy versus reference
  - Information flow directions
  - Late evaluation
- **Stackframes**
  - Static and dynamic links
  - Parameters
  - Local variables