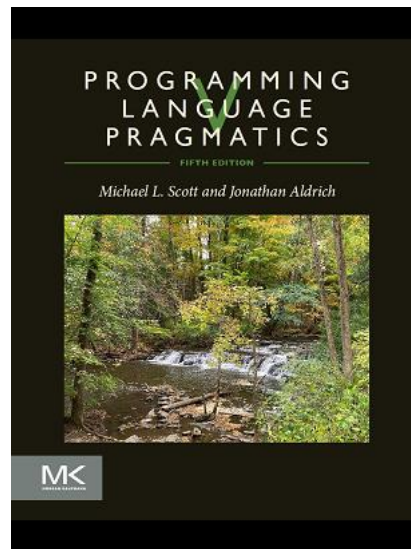


Chapter 9: Subroutines and Control Abstraction

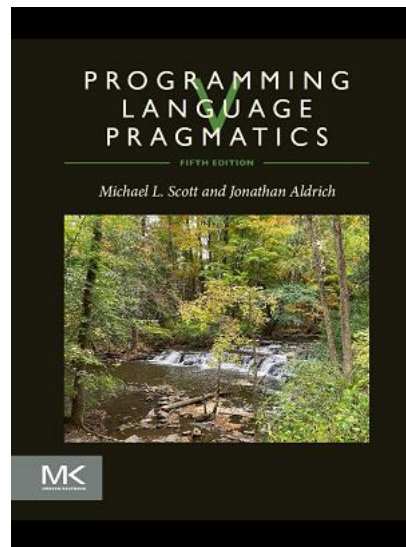


Programming Language Pragmatics, Fifth Edition
Michael L. Scott and Jonathan Aldrich

Control abstraction

- Control abstractions provide different ways of organizing program operations
 - Subroutine: performs an operation for a caller, usually accepting parameters and often returning a result
 - Exception: exits a context without returning so an error can be handled in the surrounding context
 - Coroutines: allow a program to switch between two or more execution contexts
 - Asynchronous programming: responds to sequences of events via straight-line code, doing background work between events

Section 9.2: Calling Sequences



Programming Language Pragmatics, Fifth Edition
Michael L. Scott and Jonathan Aldrich

What does a stack frame look like?

- Recall allocation strategies: static, stack, and heap
- Contents of a stack frame
 - Bookkeeping
 - return address
 - saved registers
 - static link
 - ...
 - Arguments, return values
 - Local variables
 - Temporaries

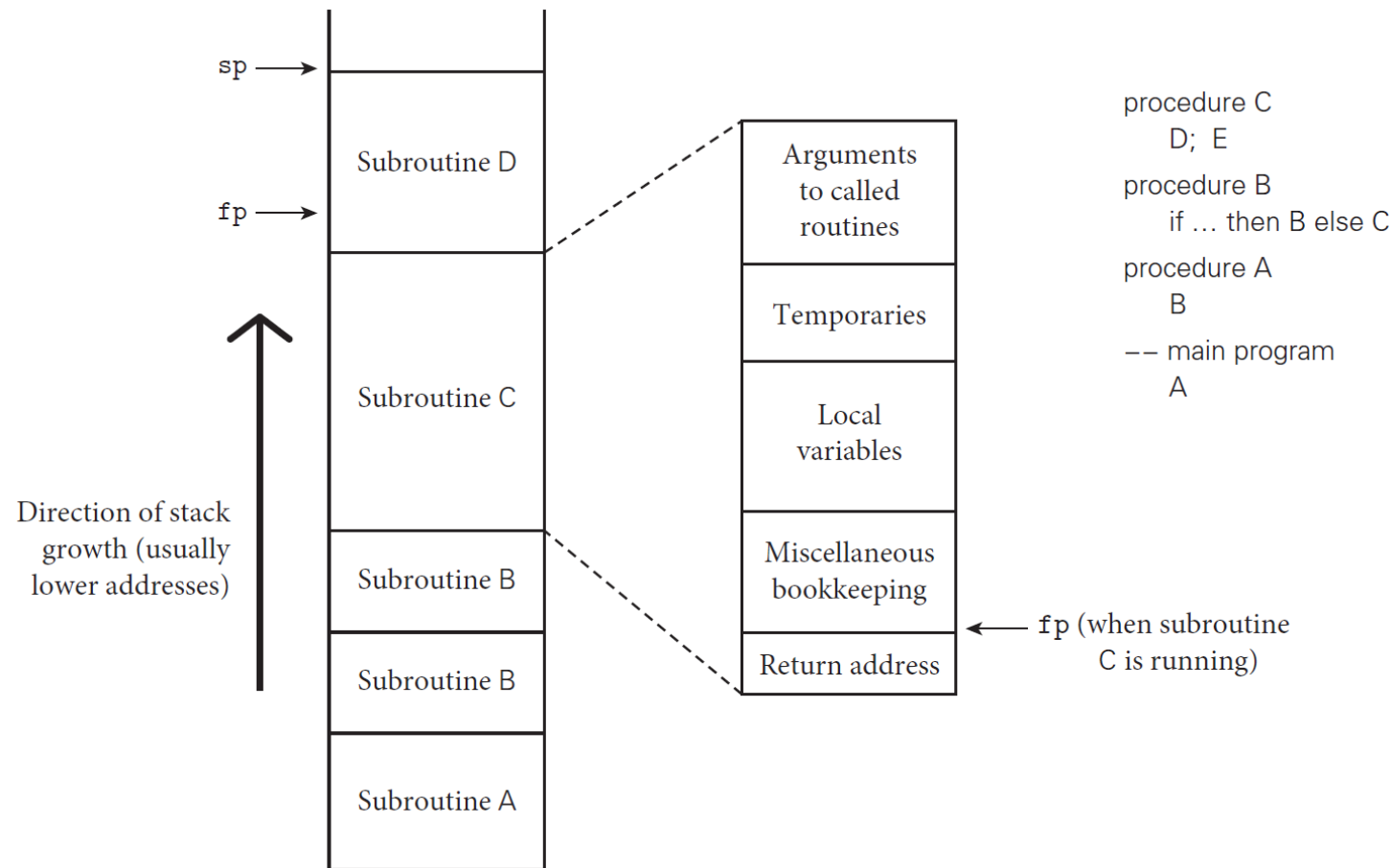


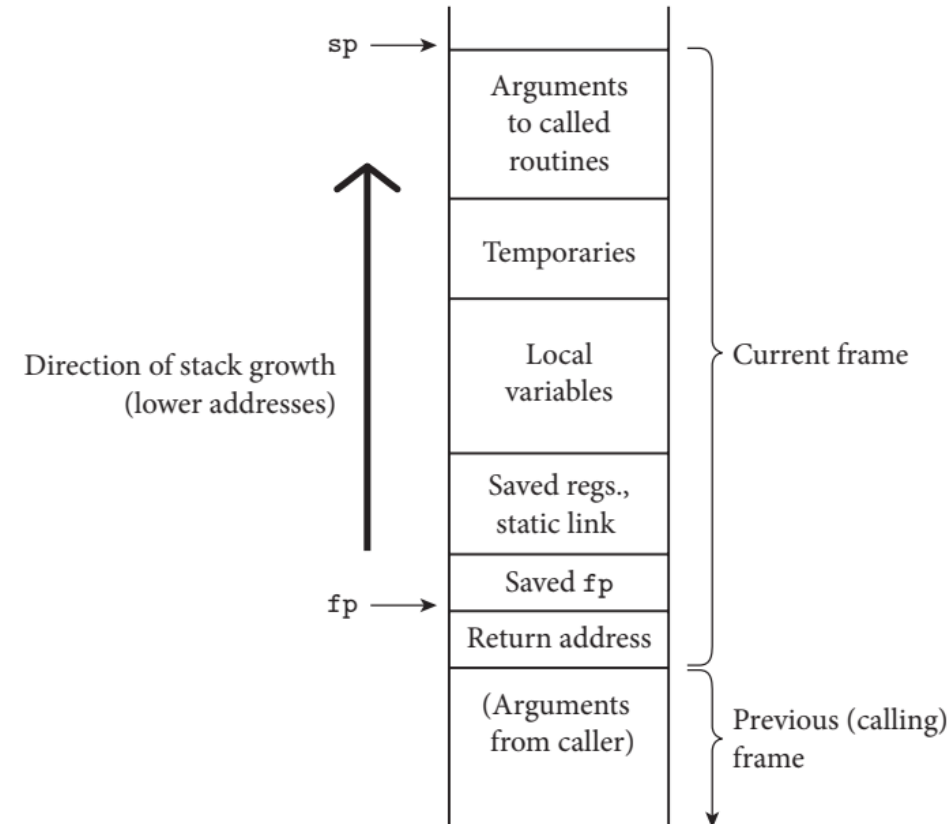
Figure 3.1

Maintenance of the stack frame

- The stack frame is maintained by
 - the *calling sequence* instructions in the caller
 - the *prologue* and *epilogue* instructions in the callee
- Space is saved by doing as much work in the callee as possible
- Time *might* be saved by doing work in the caller instead
 - The caller knows which registers are in use at the call, for example – others need not be saved
- Common strategy: divide registers into *callee-saves* and *caller-saves* sets
 - Callee-saves: stores variables & other long-lived data
 - Caller-saves: transient values that are unlikely to be needed across calls
 - Hopefully, neither caller nor callee has to save these!

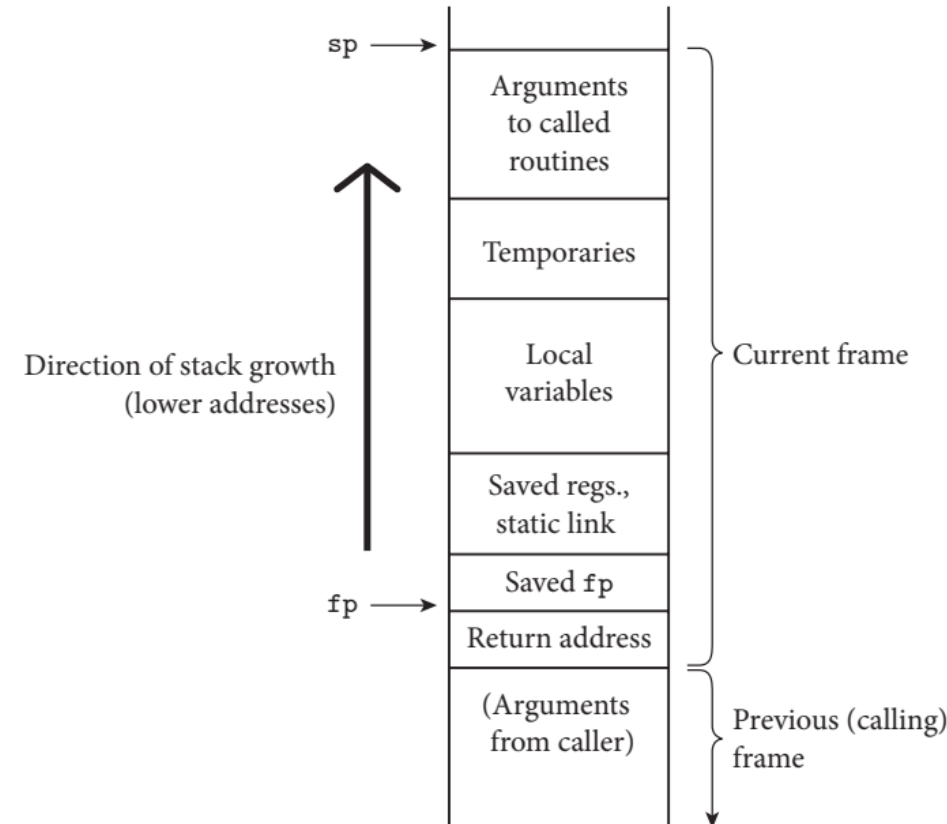
Setting up the stack frame

- The caller:
 1. saves caller-saves registers
 - if needed after the call
 2. computes arguments, moves to stack / register
 3. computes the static link, passes as argument
 - (for languages with nested subroutines)
 4. uses a subroutine call instruction
 - jumps to the subroutine
 - passes the return address on stack / register



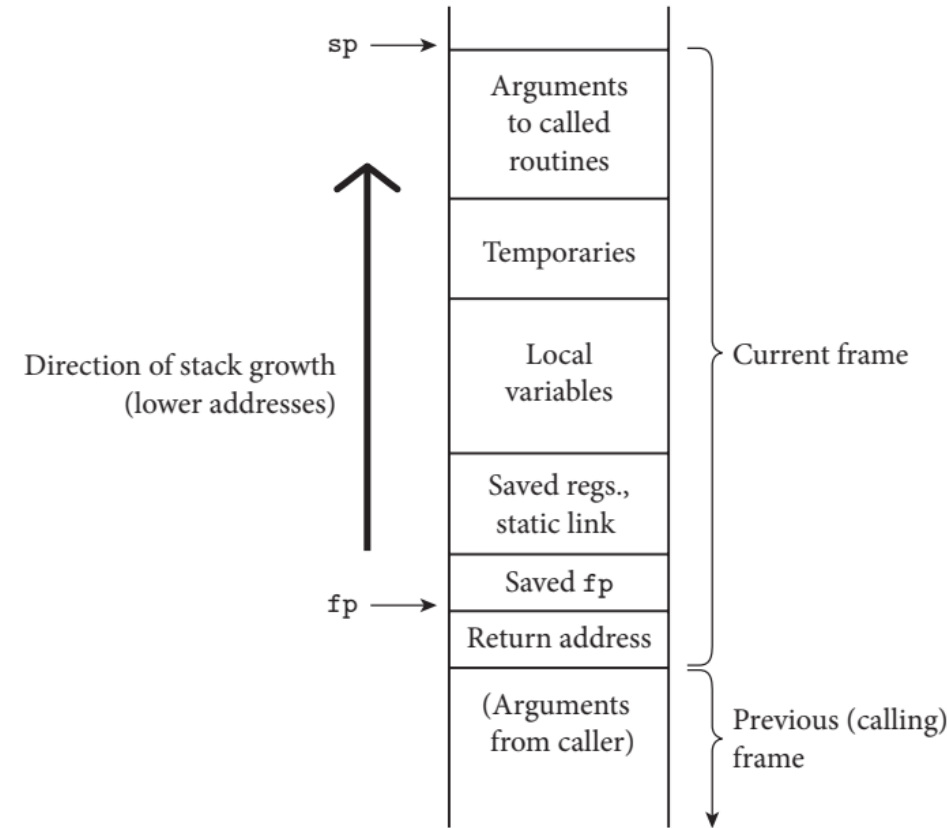
Setting up the stack frame

- The callee prologue:
 1. Allocates a frame by subtracting from stack pointer
 2. Saves old frame pointer to stack
 3. Updates frame pointer to point to frame
 4. Saves callee-saves registers
 - if they might be overwritten



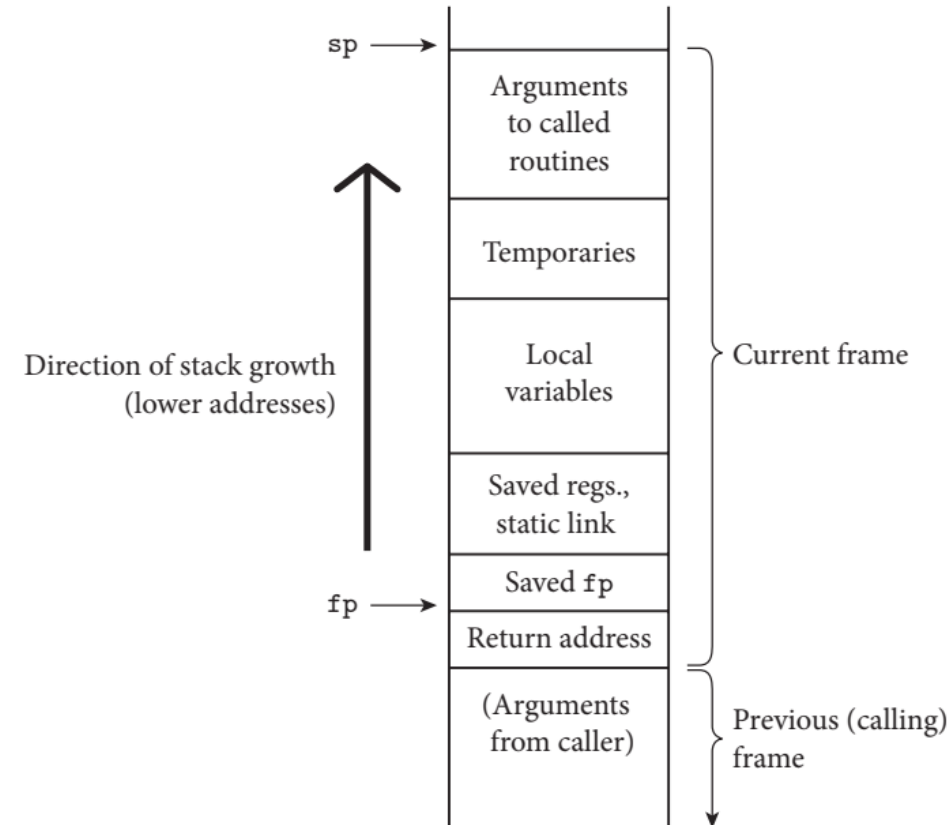
Tearing down the stack frame

- The callee epilogue:
 1. Moves the return value (if any) into a register or stack
 2. Restores callee-saves registers (if used)
 3. Restores the stack and frame pointers
 4. Jumps back to the return address



Tearing down the stack frame

- The caller:
 1. moves the return value to wherever it is needed
 2. restores caller-saves registers if needed



Modern compilers optimize stack maintenance

- Avoid special instructions, other than `call/jsr`
- Pass arguments in registers
 - space reserved on stack if needed
- Skip the frame pointer
 - works as long as the stack frame is of known (constant) size
- Implement simple leaf routines to avoid using memory at all

Calling Convention: System V AMD64 ABI

- De facto standard on Unix systems (including Intel-based Linux & macOS)
 - used for extern C calls from Rust on this platform
 - reference: <https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf>
- Callee-saved registers: rbp, rbx, r12-r15
- rsp points to the end of the latest allocated stack frame
- rsp+8 must be 16-byte aligned at a call
- you can use 128 bytes beyond (lower than) rsp and interrupts won't touch them
- the first 64-bit arguments are passed in registers, in order: rdi, rsi, rdx, rcx, r8, r9
 - additional arguments (or arguments too big for a register) are passed on the stack in reverse (right-to-left) order
- a 64-bit (or less) result is returned in rax
 - if return value is larger, space is allocated on the stack, and address is passed in rdi as a hidden first argument

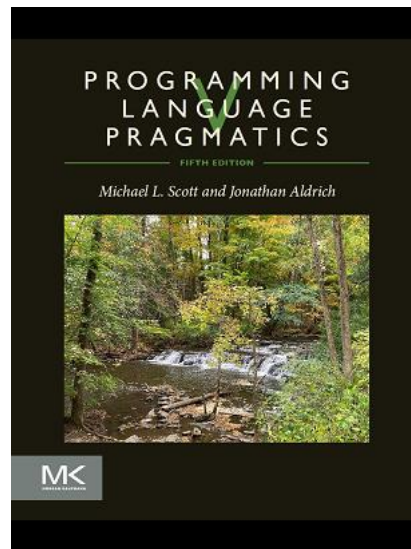
Calling Convention: System V AMD64 ABI

- *call val*
 - `push rip` - pushes (see below) `rip` (instruction pointer) register onto stack
 - `jmp val` - jumps to the provided address (literal or register)
- *ret*
 - `pop rip` - pops (see below) `rip` (instruction pointer) from the stack and continues execution
- *push val*
 - `sub rsp, 8` - decrements the stack pointer (by 8 if pushing a 64-bit register)
 - `mov [rsp], val` - writes to the space just allocated
- *pop reg*
 - `mov reg [rsp]`
 - `add rsp, 8`

Inlining

- An alternative to stack-based calling conventions is expanding the definition of a function inline in the caller
- Requires the callee to be statically known
 - E.g. no dynamic dispatch or calls through function pointers
- Generally want the callee to be short
 - otherwise inlining duplicates a lot of code
- Enables valuable optimizations
 - Eliminates calling sequence, prologue, and epilogue
 - Allows the compiler to optimize the caller and callee code together
- C/C++ have an `inline` keyword that “hints” a function should be inlined
 - In practice, compilers do what they want (and that’s usually the right thing)

Implementing first-class functions



Programming Language Pragmatics, Fifth Edition
Michael L. Scott and Jonathan Aldrich

Binding of Referencing Environments

- A *referencing environment* of a statement at run time is the set of active bindings
- A referencing environment corresponds to a collection of scopes that are examined (in order) to find a binding

First Class Functions

- Consider the following OCaml code:

```
let plus_n = fun n -> fun k -> n + k;;
let plus_3 = plus_n 3;;
let apply_to_2 f = f 2;;
```

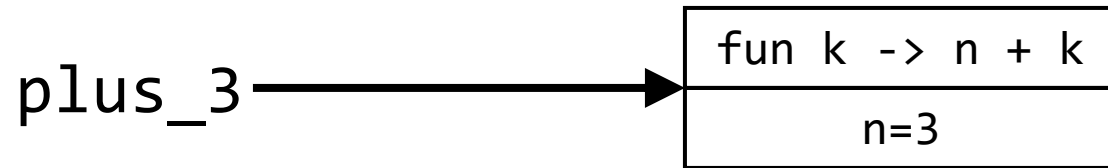
Lambda
expressions

apply_to_2 plus3 => 5

- Let's look at how this executes
(on the blackboard)

Closures

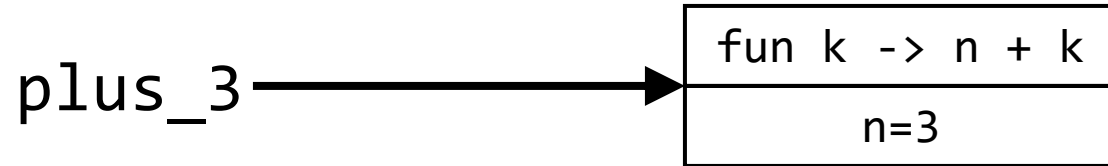
- A closure is a pair of a function and a referencing environment



- Created when a function is passed, returned, or stored
- Necessary to implement static scoping correctly
 - Otherwise the variable referenced might not be around anymore! Variable lifetime exceeds binding lifetime.
- Languages with dynamic scoping don't need them
 - Just use the caller's environment!
 - Also called "shallow binding" – closures implement "deep binding"
 - But Lisp supports closure creation if programmer asks

Closures

- A closure is a pair of a function and a referencing environment



- Several implementations
 - Allocate all referencing environments on the heap, copy a pointer into the closure
 - This is what most functional language implementations do—with optimizations when no closure will be created
 - Allocate referencing environments on the stack, copy the bindings that are used into the closure
 - This can work well if there are few captured variables and the data is immutable and small in size

Let's compile the following code using closures

let x = 3 in

let f = fn y => x + y in

f(2)

What code is generated, both for main and for the lambda body?

Let's compile the following code using closures

```
let x = 3 in
let f = fn y => x + y in
f(2)
```

ANSWER (lambda code)

```
lambda1:
    mov rax, [rdi+8]      ; load x from closure
    add rax, rsi         ; x+y
    ret                 ; return
```

ANSWER (main)

```
push 3                ; local var on stack
mov rdi, 16           ; arg to malloc
call malloc           ; allocate 16 bytes
mov [rax], lambda1   ; addr of lambda code
mov rbx, [rsp]        ; load x
mov [rax+8], rbx      ; put x in closure
mov rsi, 2            ; arg to f
mov rdi, rax          ; closure environment ptr
mov rax, [rax]        ; load address of function
call rax              ; indirect call
```

Implementing closures

- Allocating a closure to a function with code at address `addr`, with `n` closed-over variables

```
rax = allocate space of size (n+1)*8
```

```
mov [rax], addr
```

```
for (i = 1..n)
```

```
    mov [rax+i*8], var_i
```

```
// pointer to closure is in rax now
```

- Calling a closure `c`

```
mov rdi, c
```

```
// add other arguments...
```

```
mov rax, [rdi] // load the function pointer
```

```
call rax
```

- Accessing the `i`th closed-over variable inside the closure

```
mov rax, [rdi+i*8]
```

Tail Recursion

- Recursive call whose result is directly returned
- Can implement with a jump instead of a call
 - Stack frame of called function takes the place of the caller

```
int gcd (int a, int b) {  
    /* assume a, b > 0 */  
    if (a == b) return a;  
    else if (a > b) return gcd (a - b, b);  
    else return gcd (a, b - a);  
}
```