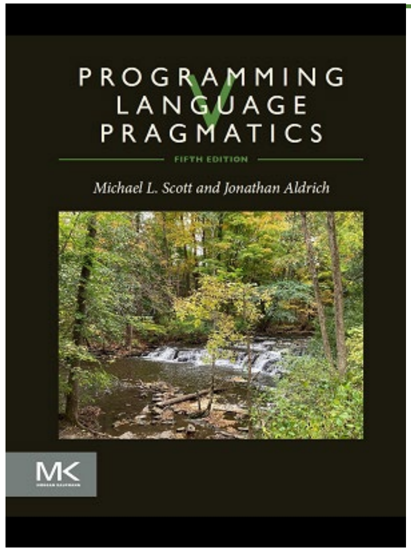
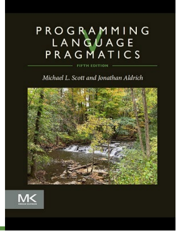
Chapter 9: Subroutines and Control Abstraction


PROGRAMMING LANGUAGE PRAGMATICS
FIFTH EDITION
Michael L. Scott and Jonathan Aldrich
MK

Programming Language Pragmatics, Fifth Edition

Michael L. Scott and Jonathan Aldrich



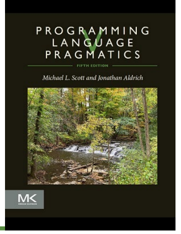
Static Scoping

- What does this Java code print?

```
class Outer {
    int x = 1;
    class Inner {
        int x = 2;
        void foo() {
            if (flag) {
                int x = 3;
            }
            System.out.println(“x = ” + x);  // what do I print?
}   }   }
```

- With static (or lexical) scope rules, a scope is defined in terms of the lexical structure of the program
    - The determination of scopes can be made by the compiler
    - Bindings for identifiers are resolved by examining code
    - Typically, the most recent binding in an enclosing scope
    - Most compiled languages, C and Pascal included, employ static scope rules

Static Scoping

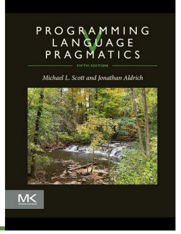


- What does this Java code print?

```
class Outer {
    int x = 1;
    class Inner {
        int x = 2;
        void foo() {
            if (flag) {
                int x = 3;
            }
            System.out.println(“x = ” + x);  // what do I print?
}   }   }
```

Answer: 2

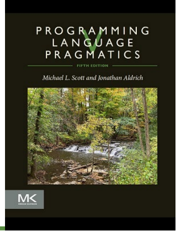
- With static (or lexical) scope rules, a scope is defined in terms of the lexical structure of the program
    - The determination of scopes can be made by the compiler
    - Bindings for identifiers are resolved by examining code
    - Typically, the most recent binding in an enclosing scope
    - Most compiled languages, C and Pascal included, employ static scope rules



Scope Rules

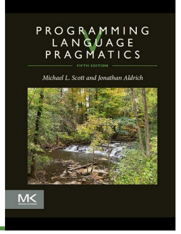
- Most closely nested rule
  - Origin: block-structured languages like Algol 60, Pascal
  - An identifier is known in the scope in which it is declared and in each enclosed scope, unless it is re-declared in an enclosed scope
  - To resolve a reference to an identifier, we examine the local scope and statically enclosing scopes until a binding is found

Dynamic Scope


PROGRAMMING LANGUAGE PRAGMATICS
Michael L. Scott and Jonathan Aldrich
MK

- (in contrast to static scope)
- No static links – just look up the latest binding of a variable in the stack
  - This may be a variable from unrelated code!
  - Makes reasoning based on program text hard

Practice with Scope Rules: Static vs. Dynamic

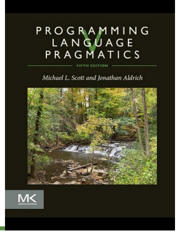

PROGRAMMING LANGUAGE PRAGMATICS
Michael L. Scott and Jonathan Aldrich
MK

```
program scopes (input, output );
  var a : integer;
  procedure first;
      begin a := 1; end;
  procedure second;
      var a : integer;
      begin first; end;
begin
  a := 2; second; print(a);
end.
```

- What is printed under static scoping?

- What is printed under dynamic scoping?

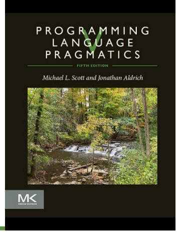
Practice with Scope Rules: Static vs. Dynamic



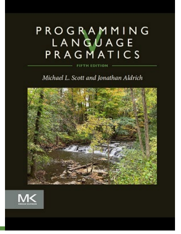
```
program scopes (input, output );
  var a : integer;
  procedure first;
       begin a := 1; end;
  procedure second;
       var a : integer;
       begin first; end;
begin
  a := 2; second; print(a);
end.
```

- What is printed under static scoping?
  - 1
- What is printed under dynamic scoping?
  - 2

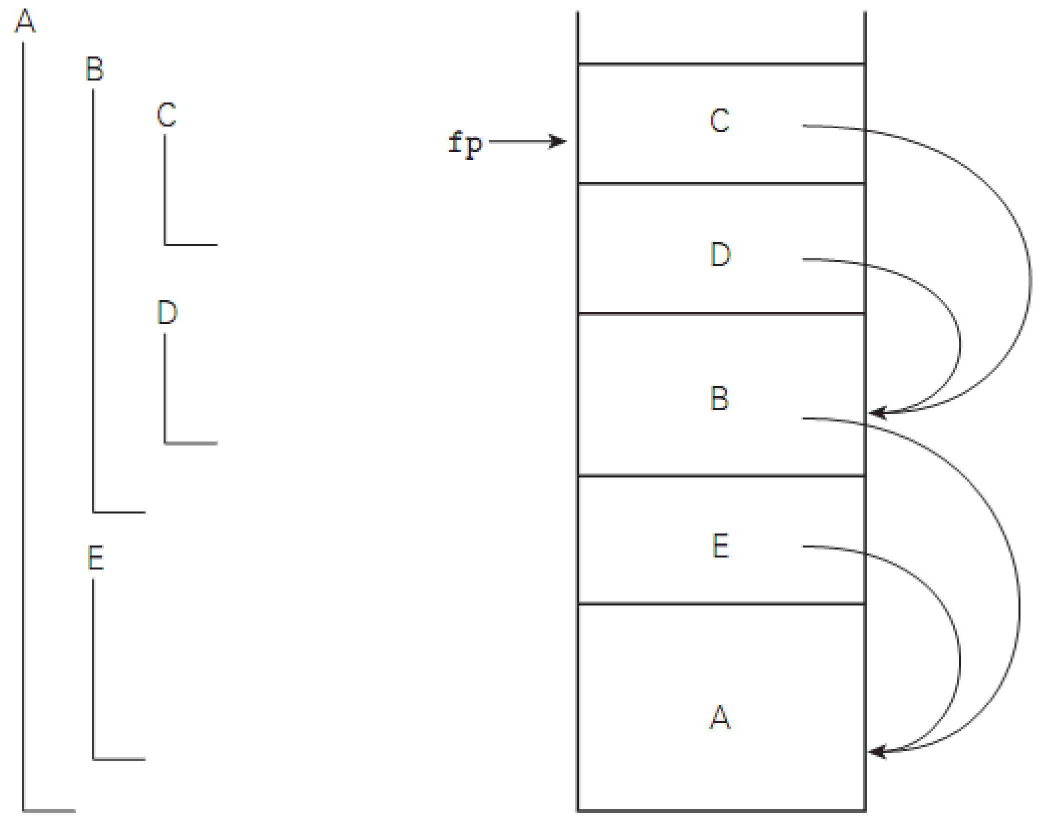
Static Links



- Access non-local variables via *static links*
  - Each frame points to the frame of the (correct instance of) the routine inside which it was declared
  - In the absence of passing functions as parameters, *correct* means closest to the top of the stack
  - You access a variable in a scope *k* levels out by following *k* static links and then using the known offset within the frame thus found



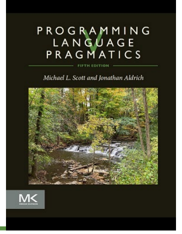
Static Chains



Q: If we are in subroutine C, what does an access to a variable v defined in subroutine A look like?

Figure 3.5 **Static chains.** Subroutines A, B, C, D, and E are nested as shown on the left. If the sequence of nested calls at run time is A, E, B, D, and C, then the static links in the stack will look as shown on the right. The code for subroutine C can find local objects at known offsets from the frame pointer. It can find local objects of the surrounding scope, B, by dereferencing its static chain once and then applying an offset. It can find local objects in B's surrounding scope, A, by dereferencing its static chain twice and then applying an offset.

Static Chains



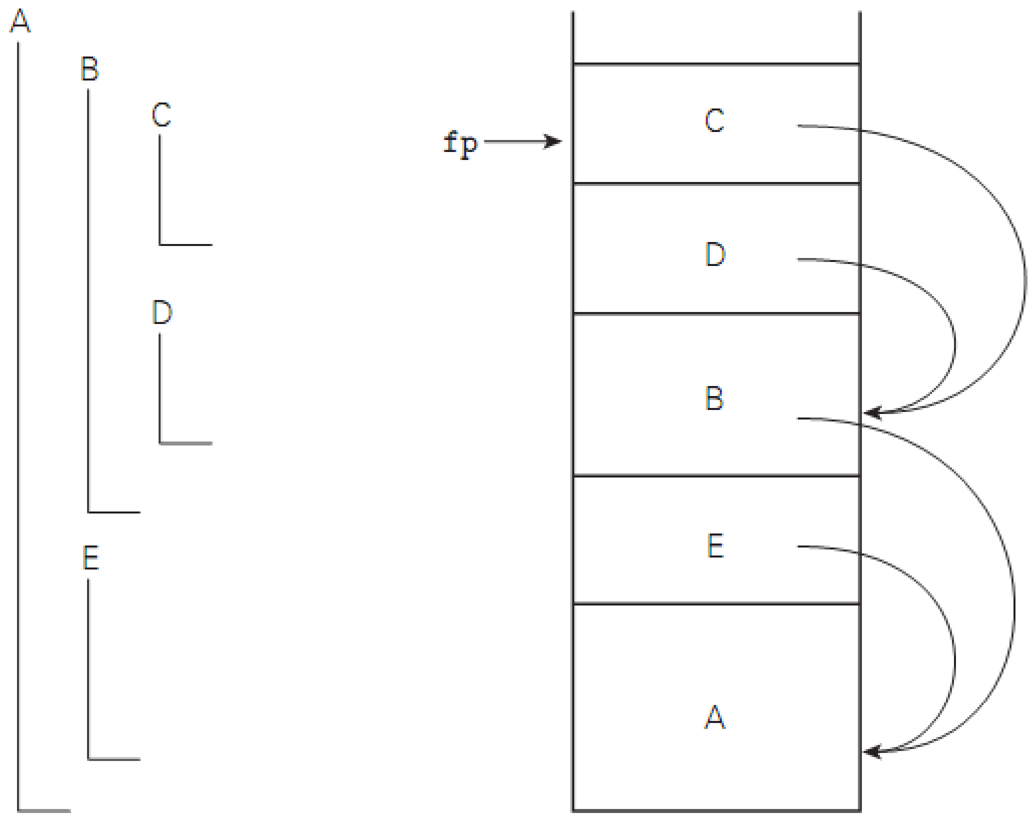


Figure 3.5 **Static chains.** Subroutines A, B, C, D, and E are nested as shown on the left. If the sequence of nested calls at run time is A, E, B, D, and C, then the static links in the stack will look as shown on the right. The code for subroutine C can find local objects at known offsets from the frame pointer. It can find local objects of the surrounding scope, B, by dereferencing its static chain once and then applying an offset. It can find local objects in B's surrounding scope, A, by dereferencing its static chain twice and then applying an offset.

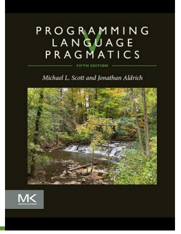
Q: If we are in subroutine C, what does an access to a variable v defined in subroutine A look like?

A: fp.link.link.v

or in assembly:

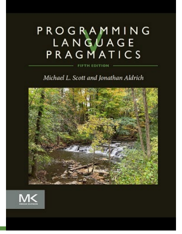
```
mov rax [rsp + link_offset]
mov rax [rax + link_offset]
mov rax [rax + v_offset]
```

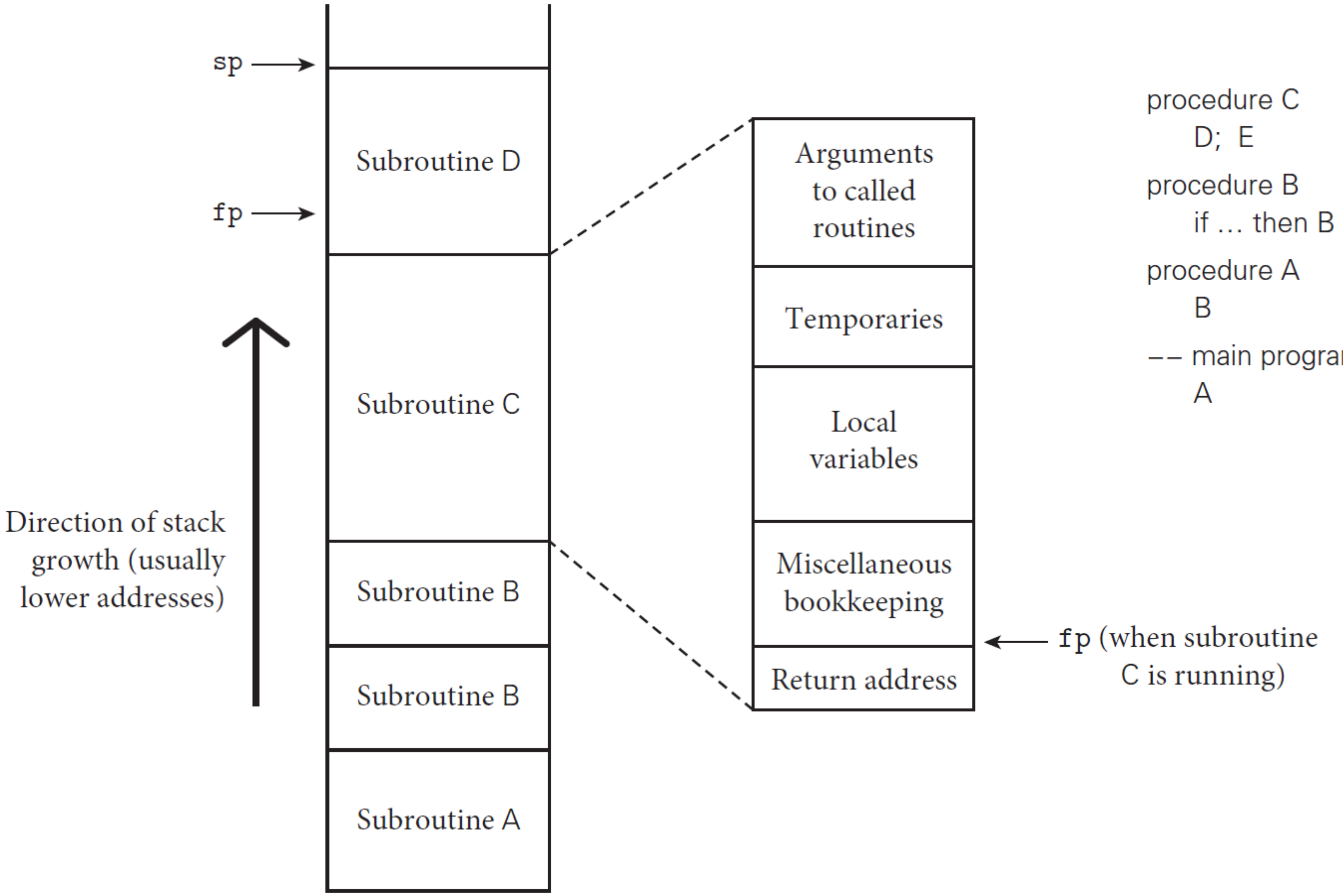
Lifetime and Storage Management


PROGRAMMING LANGUAGE PRAGMATICS
Michael L. Scott and Jonathan Aldrich
MK

- Maintenance of stack is responsibility of *calling sequence* and subroutine *prologue* and *epilogue*
  - Save space by putting as much as possible in the callee’s prologue and epilogue, rather than in the calling sequence (i.e. in the caller)…why?
    - Because most procedures have multiple callers
    - Moving a line of “administrative code” to the callee saves a line in every caller

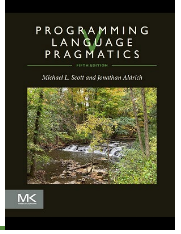
Reminder: Organization of the Stack





sp
fp
Subroutine D
Subroutine C
Subroutine B
Subroutine B
Subroutine A
Direction of stack growth (usually lower addresses)
Arguments to called routines
Temporaries
Local variables
Miscellaneous bookkeeping
Return address
fp (when subroutine C is running)

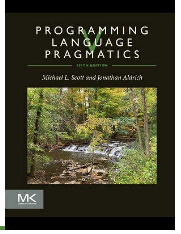
```
procedure C
    D;  E
procedure B
    if … then B else C
procedure A
    B
–– main program
    A
```



Calling Sequences

- Maintenance of stack is responsibility of *calling sequence* and *subroutine prolog* and *epilog*
  - space is saved by putting as much in the prolog and epilog as possible
  - time *may* be saved by putting stuff in the caller instead, where more information may be known
    - e.g., there may be fewer registers IN USE at the point of call than are used SOMEWHERE in the callee
- Common strategy is to divide registers into caller-saves and callee-saves sets
  - caller uses the "callee-saves" registers first
  - "caller-saves" registers if necessary
- Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time

Organization of a Stack Frame



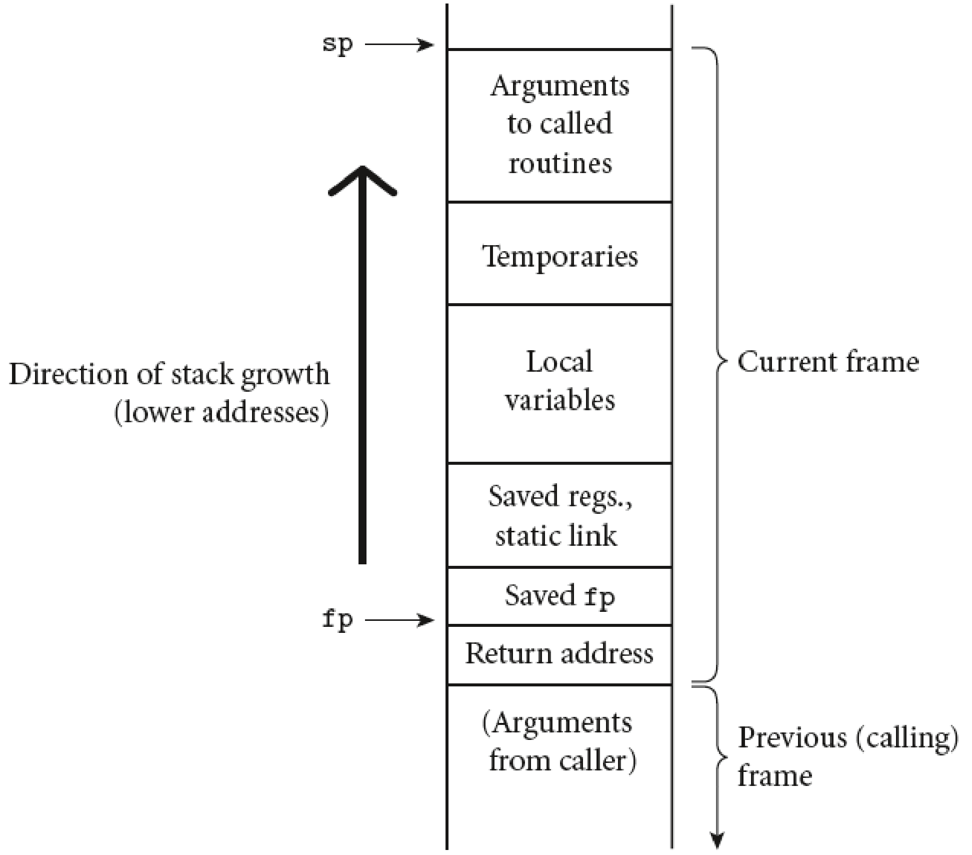
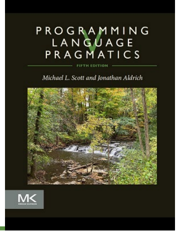


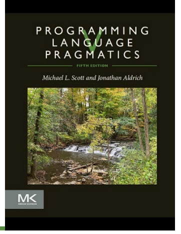
Figure 9.2 **A typical stack frame.** Though we draw it growing upward on the page, the stack actually grows downward toward lower addresses on most machines. Arguments are accessed at positive offsets from the fp. Local variables and temporaries are accessed at negative offsets from the fp. Arguments to be passed to called routines are assembled at the top of the frame, using positive offsets from the sp.

Calling Convention: System V AMD64 ABI



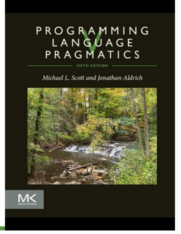
- De facto standard on Unix systems (including Linux & macOS)
    - used for extern C calls from Rust on this platform
    - reference: https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf
- Callee-saved registers: rbp, rbx, r12-r15
- rsp points to the end of the latest allocated stack frame
- 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


PROGRAMMING LANGUAGE PRAGMATICS
Michael L. Scott and Jonathan Aldrich
MK

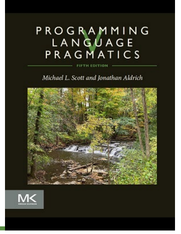
- call *val*
  - push rip - pushes (see below) rip (instruction pointer) register onto stack
  - jmp *val* - jumps to the provided address (literal or register)
- ret
  - pop rip - pops (see below) rip (instruction pointer) from the stack and continues execution
- push *val*
  - sub rsp, 8 - decrements 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

Binding of Referencing Environments


PROGRAMMING LANGUAGE PRAGMATICS
Michael L. Scott and Jonathan Aldrich
MK

- 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



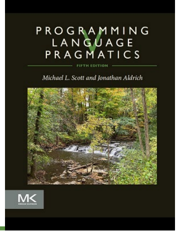
- Consider the following OCaml code:

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

apply_to_2 plus3 => 5
```

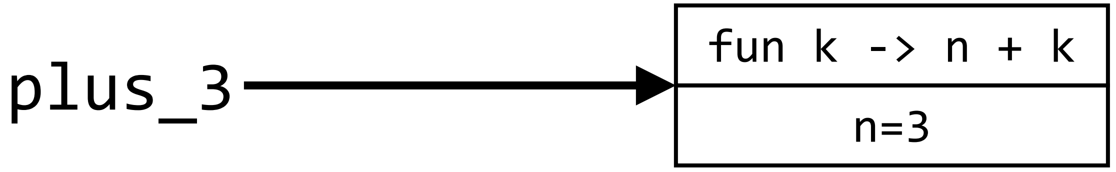
Lambda expression

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

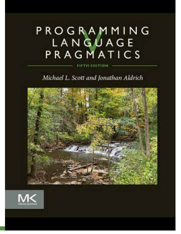


Closures

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

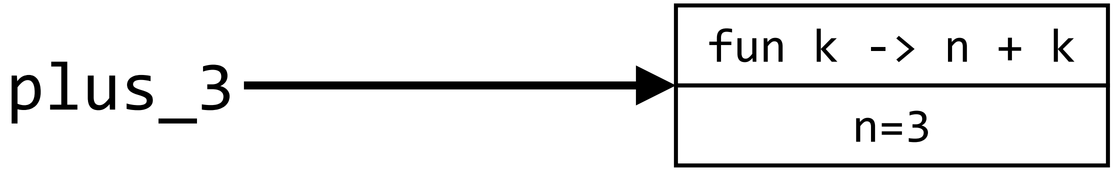


- Created when a function is passed, returned, or stored
- Necessary to implement static scoping correctly
  - Otherwise the variable referenced might not be around anymore! Variable lifetime exceeds binding lifetime.
- Languages with dynamic scoping don't need them
  - Just use the caller's environment!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



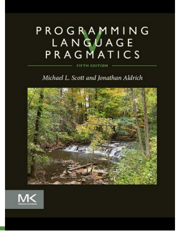
Closures

- 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

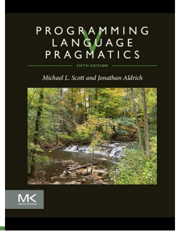
Let’s compile the following code using closures



```
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


PROGRAMMING LANGUAGE PRAGMATICS
Michael L. Scott and Jonathan Aldrich
MK

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

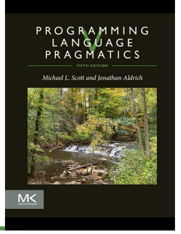
ANSWER (lambda code)

```
lambda1:
  mov rax, [rsp+8]        ; load env addr
  mov rax, [rax]          ; load x from closure
  mov rbx, [rsp+16]       ; load y
  add rax, rbx            ; x+y
  ret                     ; return
```

ANSWER (main)

```
push 3                  ; local var on stack
push 16                 ; arg to malloc
call malloc             ; allocate 16 bytes
sub rsp 8               ; pop argument off stack
mov [rax], lambda1      ; addr of lamba code
mov rbx, [rsp]          ; x
mov [rax+8], rbx
push 2                  ; arg to f
mov rbx, rax+8          ; closure environment ptr
push rbx                ; implicit closure arg
mov rax, [rax]          ; load address of function
call rax                ; indirect call
sub rsp, 16             ; pop arguments off stack
```

Implementing closures



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

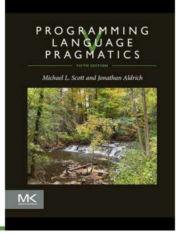
```
rax = allocate space of size (n+1)*8
mov [rax], addr
for (i = 1..n)
        mov [rax+i*8], var_i
// pointer to closure is in rax now
```

- Calling a closure c

```
mov rdi, c
// add other arguments…
mov rax, [rdi] // load the function pointer
call rax
```

- Accessing the ith 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);
}
```