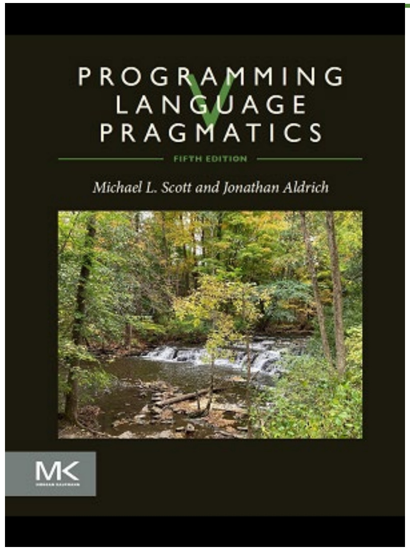
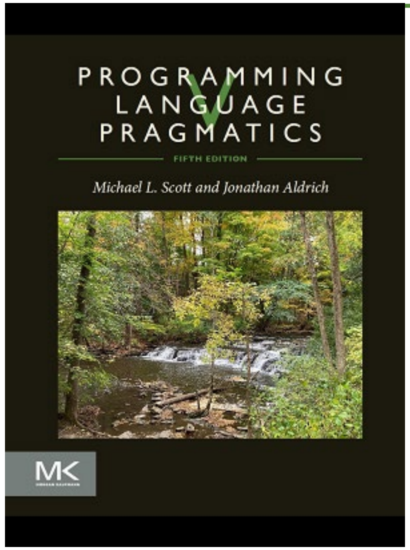
Chapter 3: Names, Scopes, and Binding



Programming Language Pragmatics, Fifth Edition

Michael L. Scott and Jonathan Aldrich

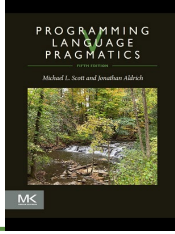
Section 3.1: Names, scopes, and binding time



Programming Language Pragmatics, Fifth Edition

Michael L. Scott and Jonathan Aldrich

Names raise the level of abstraction of programs

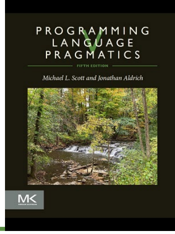


- Can refer to variables, functions, etc. using symbolic identifiers instead of addresses

- Can use a name to refer to a more complicated structure
  - A subroutine's name abstracts the implementation code
  - A class's name abstracts the data representation

- Most program data is referred to by names
  - Data on the heap is an exception—it is referred to by pointers
    - But, those pointers are stored in variables that are named!

Names, scopes, and binding

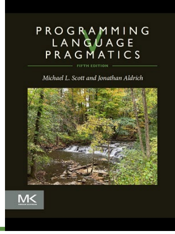


- Consider this example of a variable binding:

```
fn binding() {
    //println!("{}", name);
    let x = "Harry Q. Bovik";
    println!("Hello, {}", x);
}
```

- x is a *name*
- `let x = "Harry Q. Bovik";` is a *binding*
  - associates x with a variable
  - assigns the result of evaluating the right hand side to the variable
- The *scope* of x is where the binding is active
  - typically the statements that follow the binding

Binding time

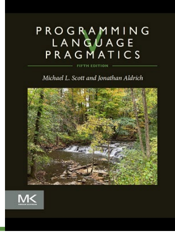


- The point at which a binding is created
  - A module import name is bound to an implementation at link time
  - A variable is bound to a value at run time

- More generally, the point at which an implementation decision is made

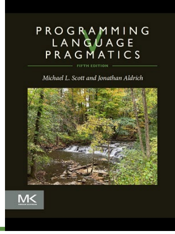
- Generally, decisions made before run-time are called *static*, decisions made at run time are *dynamic*

Decisions and their binding times

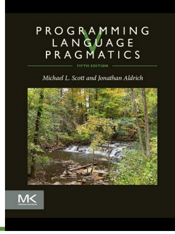


- Language design time
  - What language constructs and types are available
- Language implementation time
  - Representation and precision of primitive values, layout of the stack
    - How many bits are in a C int?
- Program writing time
  - Programmer’s choice of algorithms, data structures, and names
- Compile time
  - Mapping of source code to machine code, layout of data structures

Decisions and their binding times (continued)



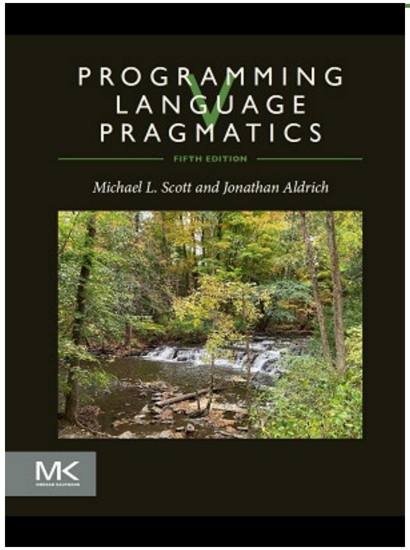
- Link time
  - Binding of a module’s imports to the referenced modules
- Load time
  - Exact layout of code in memory
- Run time
  - Binding of variables to values



Binding time in compilers and interpreters

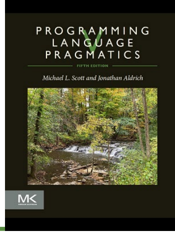
- Compilers make many decisions at compile time
  - This makes the run time more efficient, because these decisions are already made
- Interpreters delay many decisions until run time
  - Allows code to be more flexible, automatically supporting polymorphism
    - The type of data stored in each variable does not have to be determined in the source code, and can vary at run time

Section 3.2: Object lifetimes and storage management



Programming Language Pragmatics, Fifth Edition

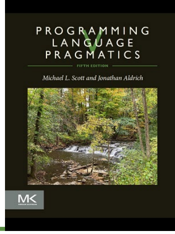
Michael L. Scott and Jonathan Aldrich



Object and binding lifetimes

- *Lifetime* of an object (e.g. a variable)
  - From when space is allocated to when it is reclaimed
- *Lifetime* of a binding (e.g. the variable's name)
  - From when it is associated with the entity to when the association ends

Q: What if the lifetime of a binding is different from the lifetime of the entity being bound?



Object and binding lifetimes

Q: What if the lifetime of a binding is different from the lifetime of the entity being bound?

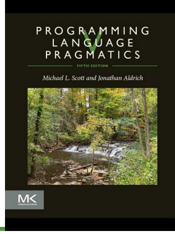
A: If binding outlives the entity, we have a *dangling reference*

- Dangling references don't usually exist as names per se, but we can create them with references

```
fn return_ptr(x:&i32) -> &i32 {
    let local = 5;
    return &local;
}
let j = return_ptr(&i);
```

Note: `rustc` will reject this program because of the dangling reference!

Object and binding lifetimes

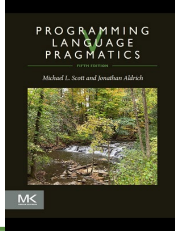


Q: What if the lifetime of a binding is different from the lifetime of the entity being bound?

A: If an entity outlives the last binding to it, we have *garbage*

- Example: in functional programming languages, a data structure may be bound to many names, and it may not be clear when the last name goes out of scope
- *Garbage collection* is used to reclaim the space used by garbage

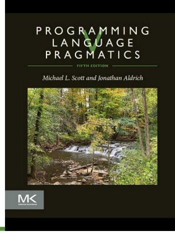
Shadowing



Q: What does this Rust code print?

```
fn shadows() {
    let x = 5;
    println!("x is {}", x);
    let x = 6;
    println!("x is {}", x);
}
```

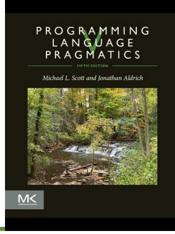
Shadowing



Q: What does this Rust code print?

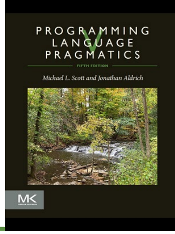
```
fn shadows() {
    let x = 5;
    println!("x is {}", x);
    let x = 6;              // shadows the earlier binding
    println!("x is {}", x); // will print 6
}
```

Deactivation of bindings

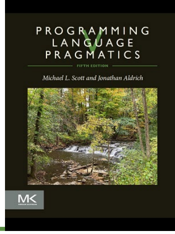


- A binding is *active* whenever it can be used
- Bindings may be (temporarily) deactivated
  - when one variable is shadowed by another with the same name
  - when calling another function, while that function executes
  - for static variables, when the containing function is not running

The timeline of an entity (e.g. a variable)

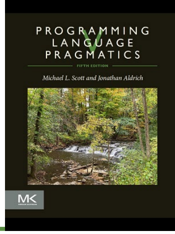


- creation of entities – e.g. at function entry, alloc stmt
- creation of bindings – at variable declaration
- use of variables (via their bindings)
- (temporary) deactivation/shadowing of bindings
- reactivation of bindings
- destruction of bindings – at end of scope
- destruction of entities – at end of scope, free stmt



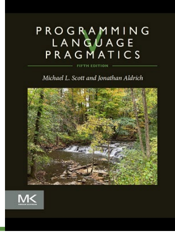
Lifetimes and storage management

- Storage Allocation mechanisms
  - Static – fixed location in program memory
  - Stack – follows call/return of functions
  - Heap – allocated at run time, independent of call structure
- Static allocation for entities that live for the entire program execution
  - code
  - globals
  - static variables
  - explicit constants (including strings, sets, etc.)
  - scalars may be stored in the instructions



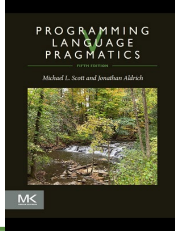
Lifetimes and storage management

- Stack allocation for entities that live for the length of a function invocation
  - parameters
  - local variables
  - temporaries
- Why a stack?
  - allocate space for recursive routines
    (not necessary in FORTRAN – no recursion)
  - reuse space (in all programming languages)

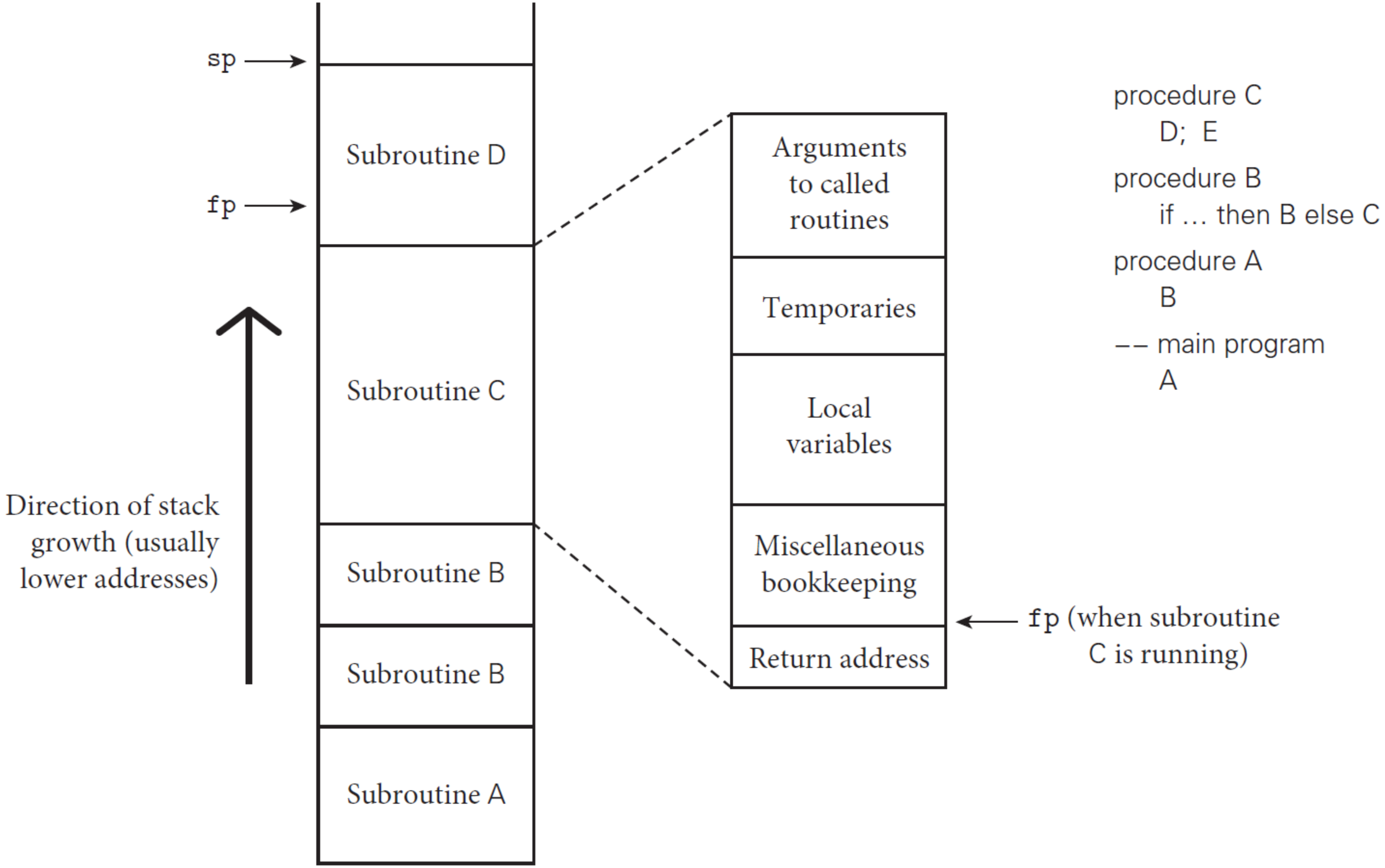


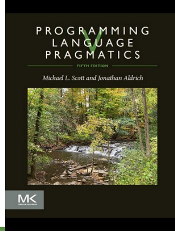
Lifetimes and storage management

- Stack allocation for
  - parameters
  - local variables
  - temporaries
- Why a stack?
  - allocate space for recursive routines
    (not necessary in FORTRAN – no recursion)
  - reuse space (in all programming languages)
- Why not a stack?
  - In functional languages, local variables may be referenced after the function returns due to *closures*, so they may be allocated on the heap



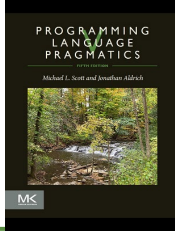
Stack-based allocation of space for subroutines





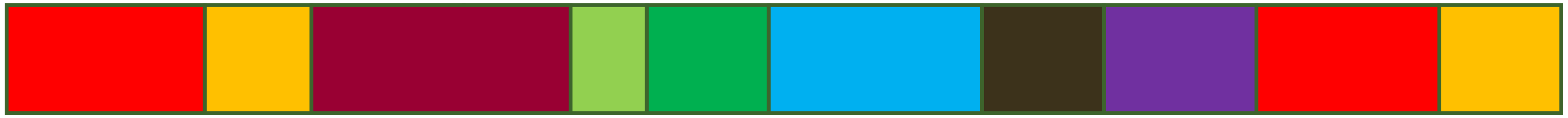
Stack-based allocation

- Maintenance of stack is responsibility of *calling sequence* and subroutine *prologue* and *epilogue*
  - Save space by doing more work in the callee’s prologue and epilogue
    - Most procedures have multiple callers



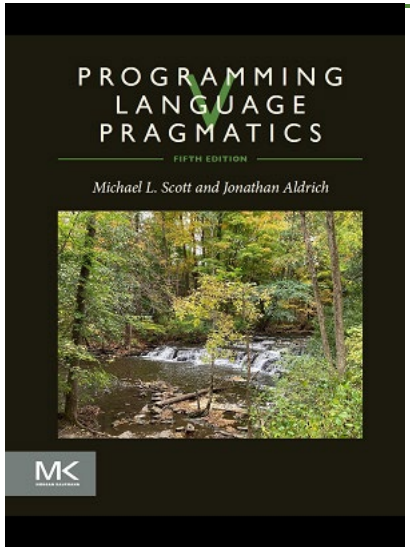
Heap-based allocation

- Heap for dynamic allocation
  - + supports lifetimes that don’t match the call stack
  - - requires explicit management or garbage collection
  - - wasted space due to fragmentation

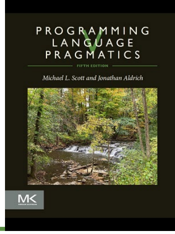


May not be able to place this block due to fragmentation, even if there is enough space overall

Stack Organization



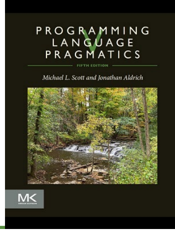
Relevant to Homework 1!



Let’s compile some code that needs the stack!

- Consider the following `snek` code: `(- 100 50)`
- For now, we want a *fully modular* compilation scheme
  - One instruction at a time, little bookkeeping keeps your life simple
    - Idea (from Monday): always leave the result in `rax` for use in the next expression
  - Production compilers will do something fancier
- Our plan
  - Compile `100`
  - Compile `50`
  - Compile the subtraction

> We have a problem! Where does the value `100` go when we are storing `50` in `rax`?
>
> We need temporary storage. Let’s use the stack.

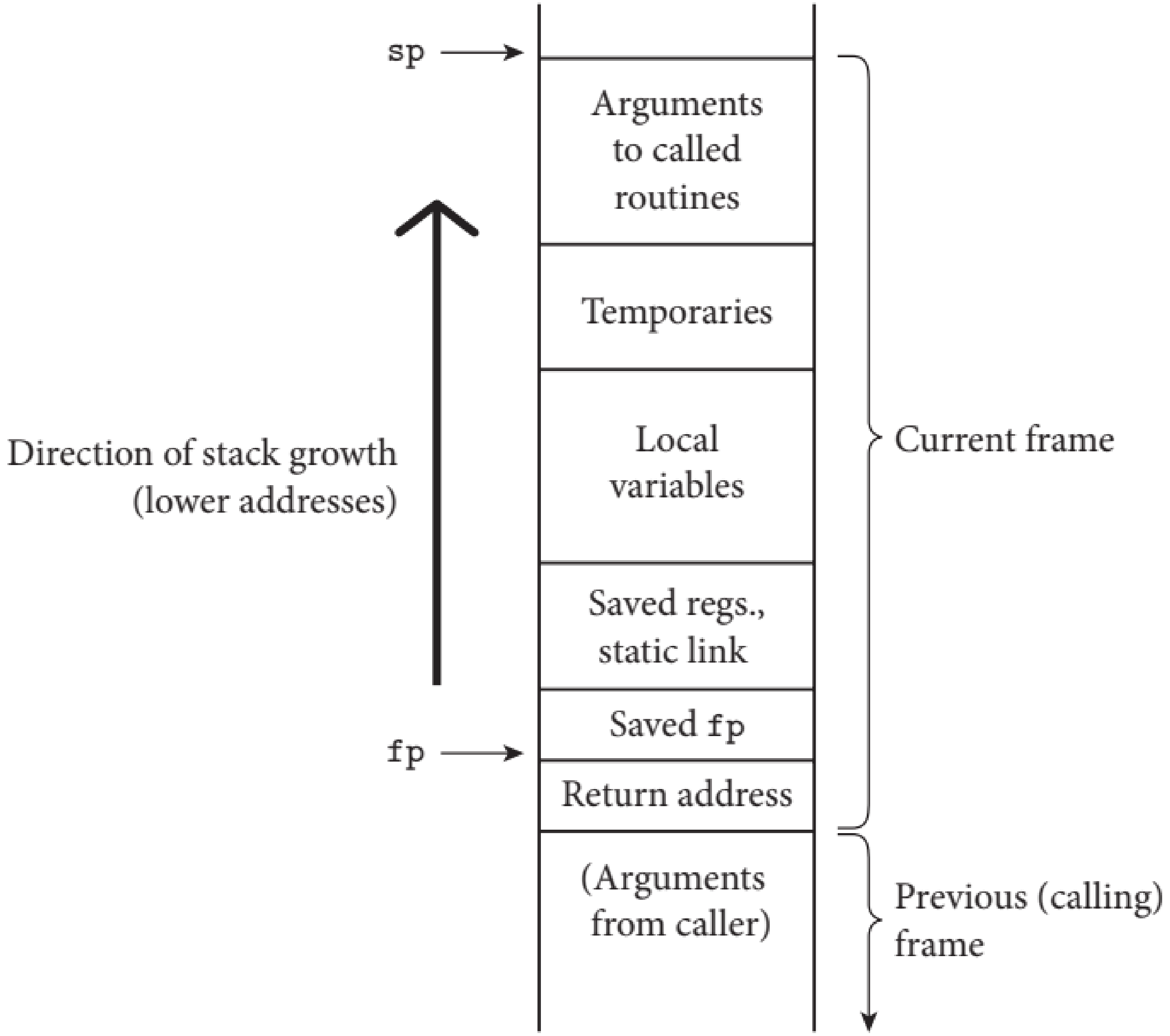


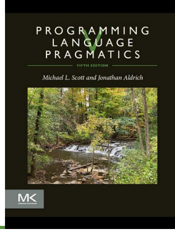
The stack frame

- The stack pointer `sp` (`rsp` in x86-64) refers to the top of stack
  - Decrement to allocate
  - `push` *`val`* allocates and writes

    ```
    sub rsp, 8          // equivalent
    mov [rsp], val      // code
    ```
  - Increment to free (often free all variables at once at the end)
  - `pop` *`reg`* reads and deallocates

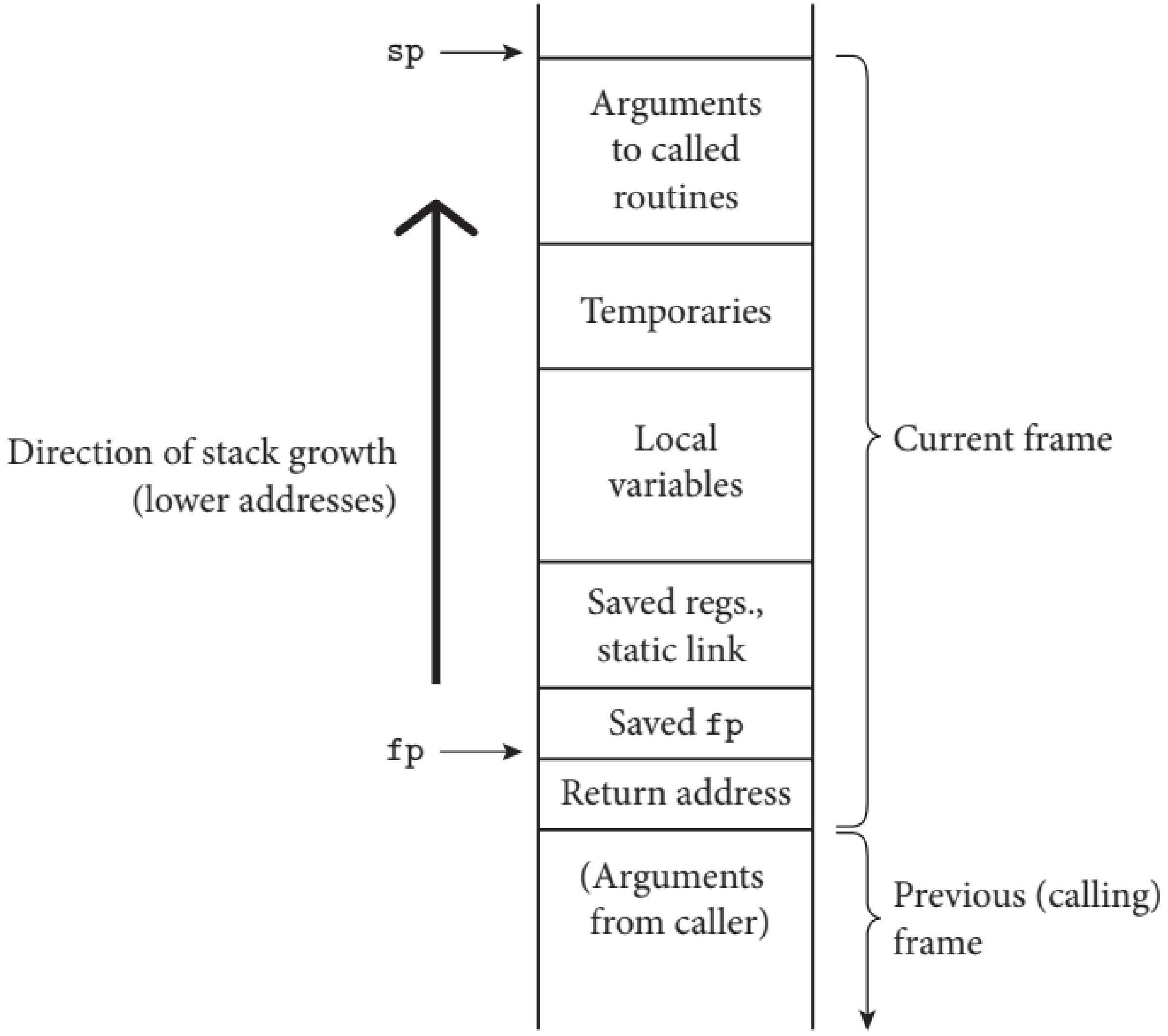
    ```
    mov reg, [rsp]      // equivalent
    add rsp, 8          // code
    ```

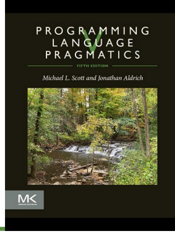




The stack frame

- The frame pointer `fp` (or *base pointer*, `rbp` in x86-64) points to the base of the frame
  - Access variables via offset
  - `mov` *`reg`*`, [rbp-n*8]`
    - Accesses the n^th variable
  - Using a frame pointer is optional! If you keep track of where `rsp` is (not hard, just bookkeeping) you can always offset from `rsp`. Modern compilers do this, then they can use `rbp` for something else.

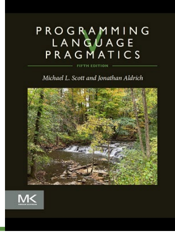




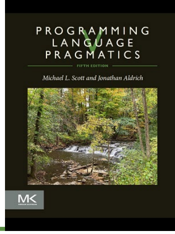
Let’s compile some code that needs the stack!

- Consider the following `snek` code: `(- 100 50)`
- For now, we want a *fully modular* compilation scheme
  - One instruction at a time, little bookkeeping keeps your life simple
    - Idea (from Monday): always leave the result in `rax` for use in the next expression
  - Production compilers will do something fancier
- Our plan
  - Compile `100`
  - Push `rax` to a temporary on the stack
  - Compile `50`
  - Move `rax` to `rbx` (since `50` is the second argument)
  - Pop the temporary back to `rax`
  - Compile the subtraction

Let’s compile some code that needs the stack!



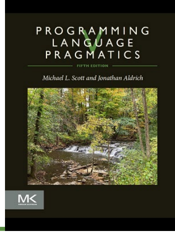
- Consider the following `snek` code: `(- 100 50)`
- Our plan
  - Compile `100` — `mov rax, 100`
  - Push `rax` to a temporary on the stack — `push rax`
  - Compile `50` — `mov rax, 50`
  - Move `rax` to `rbx` (since `50` is the second argument) — `mov rbx, rax`
  - Pop the temporary back to `rax` — `pop rax`
  - Compile the subtraction — `sub rax, rbx`
- For fun: think about how to do this better
  - The code above is far from optimal!
  - But, it is a simple translation scheme that works for Homework 1



In-class exercise

- Use the compilation scheme sketched above to compile
  `(+ 2 (- 100 50))`

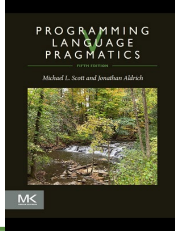
In-class exercise



- Use the compilation scheme sketched above to compile `(+ 2 (- 100 50))`
- Answer:

```
mov rax, 2
push rax
mov rax, 100
push rax
mov rax, 50
mov rbx, rax
pop rax
sub rax, rbx
mov rbx, rax            // alternative: since add is symmetric,
pop rax                 // can replace these two instructions with pop rbx
add rax, rbx
```

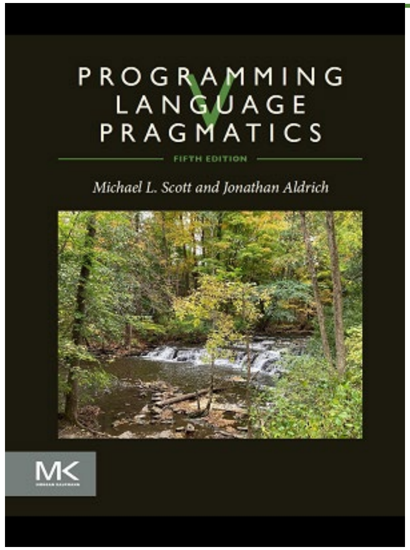
Let’s look at variables



- How to compile: `(let (x 10) (let (y 8) (+ x y)))`

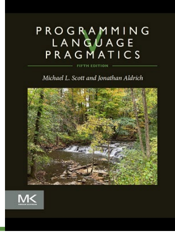
```
push rbp                // prologue: saves rbp and
mov rbp, rsp            // sets up rbp as the frame pointer
mov rax, 10
push rax                // x is at [rbp-8]
mov rax, 8
push rax                // y is at [rbp-16]
mov rax, [rbp-8]
push rax
mov rax, [rbp-16]
pop rbx                 // optimized version, uses the symmetry of +
add rax, rbx
add rsp, 16             // epilogue: deallocates variables and
pop rbp                 // restores the caller’s rbp
```

Section 3.3: Scope rules



Programming Language Pragmatics, Fifth Edition

Michael L. Scott and Jonathan Aldrich



Declarations and definitions

- Declarations
  - Introduce a name; give its type (if in a typed language)

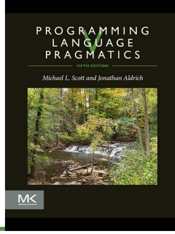
```
int x;
```

- Definitions
  - Fully define an entity
    - Specify value for variables, function body for functions

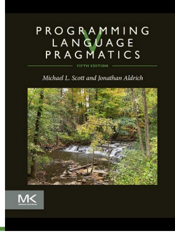
```
int x = 0;
```

- Common rules
  - Declaration before use
  - Definition before use

Rationale for ordering rules



- Declaration before use
  - Makes it possible to write a one-pass compiler
  - When you call a function, you know its signature
    - In C, this requires separating declarations from definitions to support recursion
- Definition before use
  - Avoids accessing an undefined variable
- Java relaxes both of these for classes, fields, and methods
  - But not for local variables

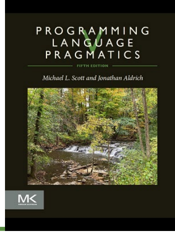


Static scoping

- Q: 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?
} }  }
```

Most recent binding of x in an enclosing scope



Static scoping rules

- 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
- “Most closely nested” rule from Algol 60
  - 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