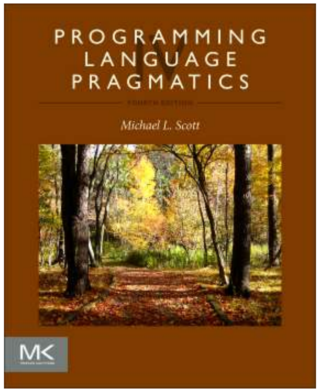
Names, Scopes, and Bindings

17-363/17-663: Programming Language Pragmatics



Reading: PLP chapter 3

Prof. Jonathan Aldrich

Name, Scope, 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

ELSEVIER

Binding

- Scope rules control bindings (of variables, functions, etc.)
  - Fundamental to all programming languages is the ability to name data, i.e., to refer to data using symbolic identifiers rather than addresses
  - Not all data is named! For example, dynamic storage in C is referenced by pointers, not names. But the pointers are ultimately stored in variables that are named.

Name, Scope, and Binding

- Some notation for scope:
  - S_2[x] indicates that x is bound in S_2

{
S_1
let x = *e*;
S_2[x]
}

- In most languages, using x in S_1 or in *e* is a compile-time error

Name, Scope, and Binding

- What happens if the scope of x is the entire block?

{

S_1[x]

let x = e[x];

S_2[x]

}

- This is true in JavaScript!
  - x will have the value **undefined** if used in S_1 or e

Name, Scope, and Binding

- In C, you can *declare* a variable without *defining* it

```
{
        S1
        int x;
        S2[x]
        x = e[x]
        S3[x]
}
```

- x is in scope in S_2, *e*, and S_3
- But if x is used in S_2 or *e*, the compiler will report a *use before initialization* warning
  - If the program is run anyway, x may have an arbitrary value (typically whatever was in the memory location being used)

Name, Scope, and Binding

- Haskell allows recursive definitions! This is OK as long as the variable being bound is used inside a function or list

let $x = e_1[x]$ **in** $e_2[x]$ -- *general form*
let x = x **in** x+1 -- *run time error (black hole)*
let x = 1 : x **in** ... -- *OK: x is a cyclic list of 1s*

let f = \n -> **if** n == 1 **then** 1 **else** n * f(n-1) **in** … : x
-- *OK: defines factorial*

Lifetime and Storage Management

- *Lifetime* of an entity (e.g. 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

```
fn return_ptr(x:&i32) -> &i32 {
    return x;
}
fn main() {
    let i = 1;
    let j = return_ptr(&i);
    println!("j is {}", j);
    // j is not used past here ...
}
```

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

Lifetime and Storage Management

- *Lifetime* of an entity (e.g. 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
  - 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 pointers

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

ELSEVIER

Lifetime and Storage Management

- *Lifetime* of an entity (e.g. 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
  - If binding ends before the entity, we have *garbage*
    - Can happen in functional languages

let f(x) =
        let y = x + 1 **in**
        fn z => y + z *// have to keep y around when f returns*
in let g = f(1) *// y is used in the returned function g*
in let h = g(2) **in**
… *// at this point y is garbage*

Lifetime and Storage Management

- Here’s the Rust version

```
fn return_closure() -> Box<dyn Fn(i32) -> i32> {
    let increment = 1;
    let f = move |x| x+increment;
    return Box::new(f);
}
    let f = return_closure();
    let k = (*f)(1);
```

Lifetime and Storage Management

- *Lifetime* of an entity (e.g. 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
  - If binding outlives the entity, we have a *dangling reference*
  - If binding ends before the entity, we have *garbage*
- A binding is *active* whenever it can be used
- A *scope* is the largest program region where no bindings are changed
  - Typically from a variable's declaration to the end of a block

ELSEVIER

Lifetime and Storage Management

- What does this Rust code print?

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

Lifetime and Storage Management

- Bindings may be (temporarily) deactivated
  - When one variable is *shadowed* by another with the same name

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

  - When calling another function, while that function executes
  - For static variables, when the containing function is not running

Lifetime and Storage Management

- Typical timeline (e.g. for variables)
  - 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

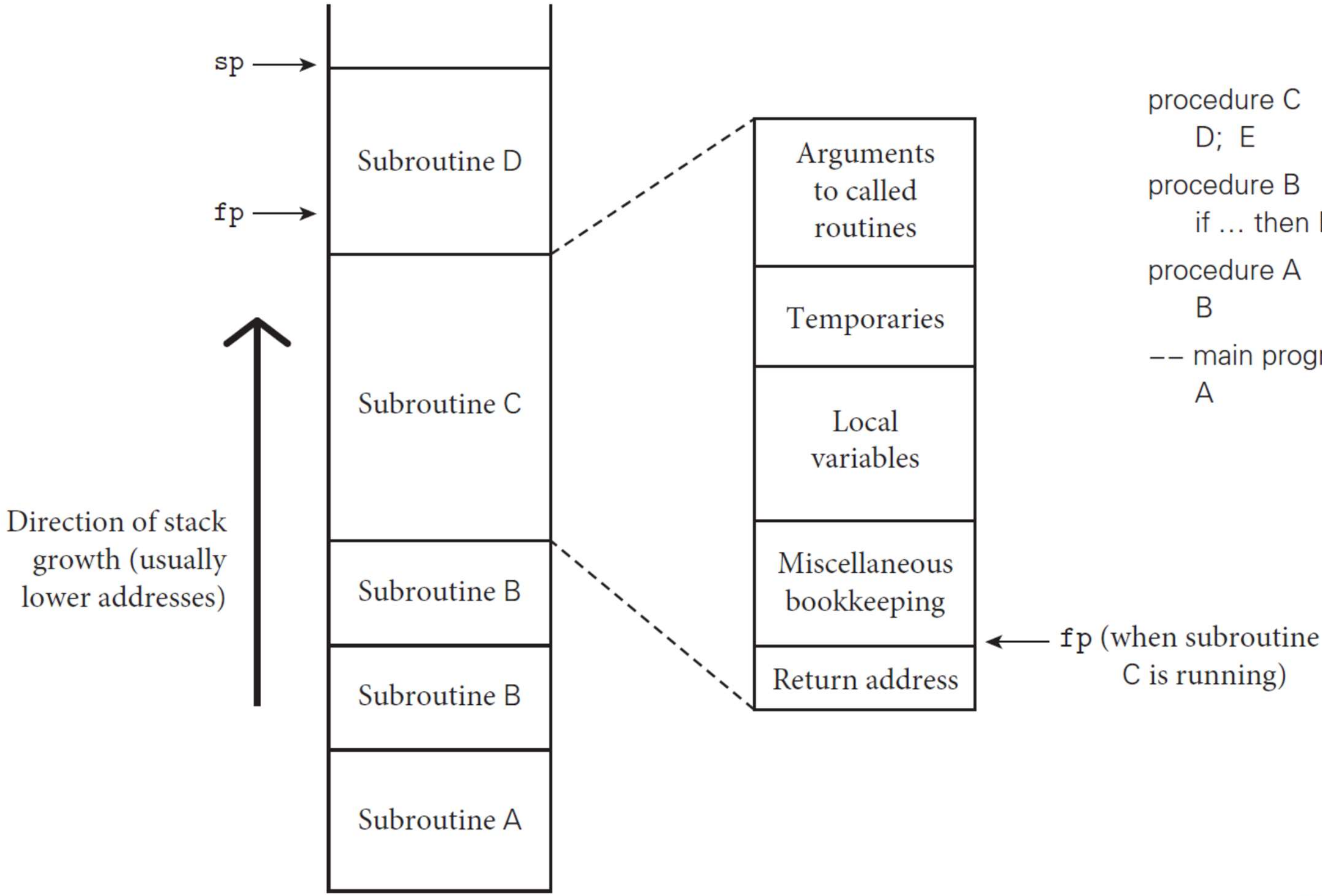
Lifetime and Storage Management

- 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
  - –code
  - –globals
  - –static variables
  - –explicit constants (including strings, sets, etc.)
  - –scalars may be stored in the instructions

Lifetime 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?
  - We already saw that *closures* can be an exception

Lifetime and Storage Management




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

Lifetime and Storage Management

- Let’s look at compiling some Snake code
  - Next week’s homework

(- 100 50)

(+ 2 (- 100 50))

Slight change from the original question in class, illustrates temporaries on the stack better

(let (x 10) (let (y 10) (+ x y)))

ELSEVIER

Lifetime and Storage Management

Answer: compiling (- 100 50)

```
mov rax, 100
mov [rbp-8], rax
mov rax, 50
mov rbx, rax
mov rax, [rbp-8]
sub rax, rbx
```

Compilation of 100

Compilation of 50

Compilation of subtraction

Note: the subtraction code above is slightly different from the solution in class; the code given here is slightly longer but is more robust because it preserves evaluation order and so will continue to work in an imperative setting.

Lifetime and Storage Management

Answer: compiling (+ 2 (- 100 50))

```
mov rax, 2
mov [rbp-8], rax
mov rax, 100
mov [rbp-16], rax
mov rax, 50
mov rbx, rax
mov rax, [rbp-16]
sub rax, rbx
add rax, [rbp-8]
```

Lifetime and Storage Management

Answer: compiling (let (x 10) (let (y 10) (+ x y)))

```
mov rax, 10
mov [rbp-8], rax        ; x
mov rax, 10
mov [rbp-16], rax       ; y
mov rax, [rbp-8]
mov [rbp-24], rax       ; temporary
mov rax, [rbp-16]
add rax, [rbp-24]
```