Names, scope, and binding

nprenet@bsu.edu

CS431 - Spring 2022

Names allow us to refer to

- variables, constants, operations, types, ... on using symbolic identifiers instead of low-level constructs (addresses).
- abstraction of complicated program fragments: functions

It is binding time

A **binding** is a an association between a name and the thing it names.

Static vs. Dynamic

1. Before runtime, or **static binding**
    a. Language design time. E.g. constructors, control-flow construct, primitive types
    b. Language implementation time. E.g. I/O (OS-dependent)
    c. Program writing time. E.g. named custom data structures
    d. Compile time: mapping of high-level names to machine code, including layout of data in (virtual) memory
    e. Link time. E.g. modules imports
2. At runtime, or **dynamic binding**

A broad term that covers the entire span of the program's execution from the beginning to the end. Contrast:

- **compiler-based languages**: more efficient because they make earlier decisions about binding, including memory layout.
- **interpreters** that must analyze the declarations every time the program runs: less efficient, more more flexible.

A number of languages combine both approaches. E.g. C# that relies both on traditional compilation, and just-in-time (JIT) compilation by the runtime virtual machine;[1] compiled languages that have dynamical binding and postpone a number of decisions on memory layout until runtime (Python).

Object lifetime and storage

Distinguish between

- the **object lifetime**: between the creation of an object and its destruction
- the **binding lifetime**: between creation of a name-object binding and its destruction.

Object lifetime and binding lifetime may not overlap. E.g. dangling reference–a programmer's error. E.g. references passed as parameters to subroutines, whose lifespan is generally much shorter than the object they refer to.

[1] The Common Language Runtime aka. CLR

Three storage allocation mechanisms: static, stack, heap.

Static objects

Static objects are given an absolute address, that is retained throughout the program execution. E.g. global variables, numeric and string literals, debugging tables, compile-time named constants ($\neq$ elaboration-time constants that are assigned at runtime, even if they are not mutable).[2]

Stack objects

They are allocated and deallocated in LIFO order, with subroutines calls: in any language that allow recursion, where the number of local variable instances may be infinite.

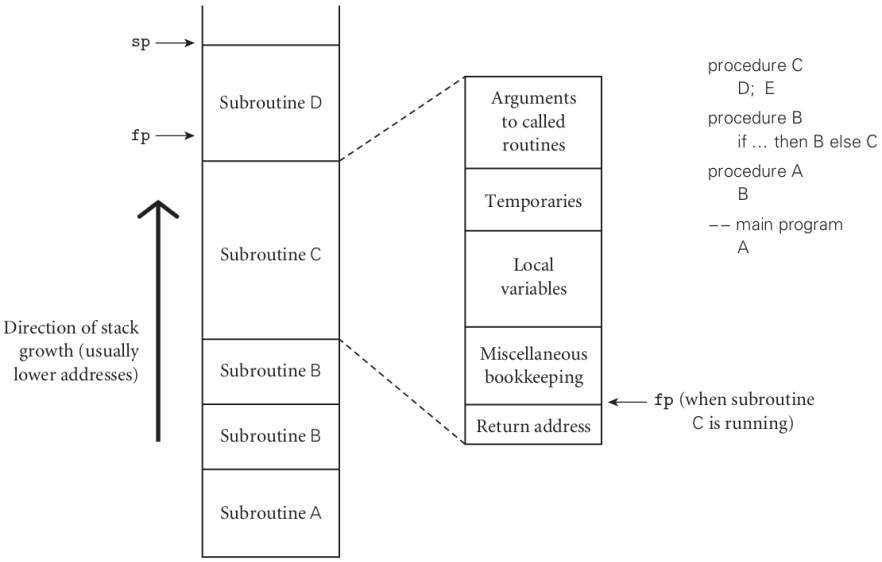


Figure 1: Stack records

Heap-based allocation

A **heap** is a region of storage in which subblocks can be allocated an deallocated at arbitrary times. E.g. dynamic data structures (linked), general strings, lists, sets (all mutable containers)

- commonly implemented as a single linked list of heap blocks not yet in use, that is fragmented further with each allocation of a new object (see Figure~2);
- deallocation is more efficient when done by hand, but error-prone, hence the **garbage collection algorithms**, that automate the task, at the expense of performance.

Scope rules

Scope: the textual region of the program in which a binding is active.

In most modern language, scope of a binding is determined statically at compile time, based on purely textual rules (hence the name: **lexical scope**). Which languages construct create a new scope?

[2]With no recursion, variables local to a subroutine could be allocated statically (one instance for call of the same function).

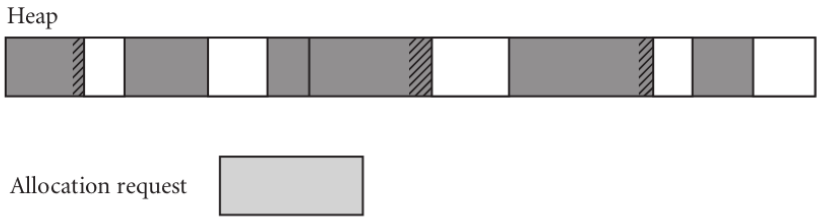


Figure 2: Heap allocation

- (most languages) a subroutine introduces a new scope; any binding of the same within the subroutine block hides existing binding of the same name in the surrounding scope; when exiting the subroutine, local bindings are destroyed, and existing bindings of the outer scope are restored.
- (most languages) body of a module, class
- (some languages: C and family) structured control-flow statements (`for ...`, `while ...`)
- specific constructs whose only effect is to create a new scope: any `{...}` block in C, `let ... in ...` in Haskell, `(let ... ())` in Scheme. E.g In the following Javascript function, the `{ let ... }` syntax delimits the scope for `x`'s binding:

```
(function(){
   var x = 1;
   { let x = 2 }
   console.log(x)
})()                // 1
```

Static scope rules

The set of all bindings that are active in any point of the program's execution is the **referencing environment**, as determined by **scope rules**: the compiler determines the current bindings by finding a matching declaration in the closest surrounding block; if not found, in the next closest surrounding scope (closest nested scope rule, as applied, f.i. to nested subroutines).

E.g. In the code below (`static_scope.js`), function `sub1` is *declared* in a block where `x` is bound to 3.

```
(function (){
  function sub1(){
    var x = 7;  // local
    sub2();
  }
  function sub2(){
    console.log(x)
  }
  var x = 3;
  sub1();
})()           // 3 (not 7)
```

Many variations: Should local binding be declared at the top of the block (methods)? Can they be used before they are declared (Java/C#: yes for class members, but not for method locals)? Is the binding valid throughout the block (C#, Python, Javascript), and only after the declaration (Java)?

Rules that determine access to the **global scope** depend on the language. E.g. In Javascript any variable that is *not* declared with a `var` qualifier exists in the global scope.

```
x = 1;
(function(){
   x = 2;          // global
   var y = 3;      // local
})()               // calling the anonymous function

console.log(x); // 2
console.log(y); // Uncaught ReferenceError: y is not defined
```

Example: Python's cope rules

By default, **global variables** are *not* visible from a subroutine's scope. The code below raises an exception:

```
x = 1
def func:
  print(x)
func()     # UnboundLocalError: local variable 'x' referenced before assignment
```

An assignment of `x` just creates a local binding:

```
x = 1
def func:
  x = 2
  print(x)
func()     # 2
print(x)   # 1
```

The **`global` qualifier** allows for a read-write access to the global scope:

```
x = 1
def func:
  global x
  x = 2
func()     # 2
print(x)   # 2
```

(function a(){ var x = 1 do { var x = 2 } while (false) console.log(x) })()

Loop constructs do *not* create a new scope in Python![3] **Only functions** do. In the code below, both the scopes of both `i` and `x` spans from their assignment/declaration to the end of the function's body:

```
def func():
  for i in [1,2,3]
    x = i
  print(i, x)
func()       # 3, 3
```

Variables that are in the outer scope of a nested function are read-only: by default, any assignment of a variable with the same name creates a new binding that shadows the first one. In the program below, the name `x` is bound to two different variables, at lines 2 and 4, respectively:

```
def a():
    x = 1
    def b():
        x = 5    # a local binding, not a new assignment
        print(x)
    b()
```

[3] In contrast with C or Java/C#.

```
    print(x)
a()     # 5, 1
```

At line 8, the referencing environment only contains the original binding defined at line 2, where `x` is 1. The default behavior can be overriden with the **`nonlocal` keyword**. Compare with the foollowing program, where the assignment in routine `b()` relies on the binding defined at line 2:

```
def a():
    x = 1
    def b():
        nonlocal x
        x = 5     # uses and updates binding in outer scope
        print(x)
    b()
    print(x)
a()     # 5, 5
```

Dynamic scope rules

With dynamic scope rules, the referencing environment for a function is determined by the scope in which the function *executes*. Under the assumption of dynamic scope rules, what would this Javascript code do?

```
function big(){
 function sub1(){
  var x = 7;  // local
  sub2();
 }
 function sub2(){ console.log(x) }
 var x = 3;
 sub1();
}
big();    // 7
```

Under the dynamic scope assumption, the referencing environment for routine `sub2()` is determined by the `sub1()` routine's local scope, where `x` is bound to 7. Such rules are rarely used in programming languages, because they make it very hard to reason about the correctness of programs: it is not immediately clear in line 4 that routine `sub2()` uses the `x` variable!

Modules

Goal: hide some bindings from the current environment, for the sake of data abstraction.

- narrow interfaces
- encapsulate collections of routines, with their surrouding scope and bindings: variables, types, etc.
- export/import exposes only the bindings that are needed by the main program

Binding of referencing environments: closure example

In static scoping the referencing environment is determined by static scoping rules (based on the lexical nesting of blocks where variables are declared). Now what happens in languages that allow one to create a *reference* to a subroutine, f.i. by passing it as a parameter? Should bindings be searched

- in the scope(s) surrounding the function *definition*? → **shallow binding** (commonly associated with dynamic scope)
- in the referencing environment of the function *call*? → **deep binding**

Deep binding allows for closures, as illustrated by the two example below.

Example 1. Deep binding in Python

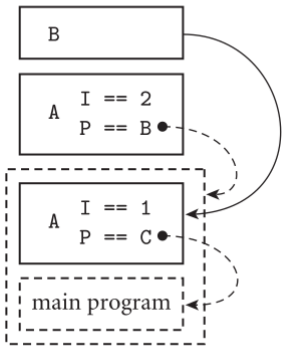
```
def A(I, P):

    def B():
        print(I)

    # body of A:
    if I > 1:
        P()
    else:
        A(2, B)

def C():
    pass    # do nothing

A(1, C)     # main program
```



When B is called via formal parameter P, two instances of I exist in the runtime stack. Because the closure or P was created in the initial invocation of A, B's static link points to the frame of that earlier invocation. Accordingly, the output is 1, not 2.

Example 2. A Python closure

A **closure** is a bundle of

- a reference to a subroutine
- an explicit representation of the subroutine's referencing env. in which the routine would execute if called at the time the closure is created

```
def sub1():
    a = 1
    def sub2():
        a = 2
        def sub3():
            print(a)
        return sub3
    return sub2()

a = 3
f = sub1()
f()         # 2
```

When function `sub3` is returned from `sub2`, and then, in turn, from `sub1`, it "carries" with its its referencing environment! When it executes (`f()`), the function has an access to the scope that surrounds its definition (were `a=2`). More rigorously: when the code passes the routine as a parameter, it also *binds the current environment*, that is then restored when the routine is finally called.

Do you need closures, when you have classes?

In **functional languages** that not rely on assignment and state, closures are a natural way to capture the context of a function, to be used at a later stage of the computation. In Haskell, partially applied functions (or sections) are closures, as illustrated below:

```
g x y = x + y

f = (g 2) -- closure!
```

```
f 1 -- 3
```

By contrast, in **object-oriented languages** like C# and Java, there is a lesser need for closures: static scoping is enough, and classes precisely allow to access methods with their context. To pass a subroutine with context, we encapsulate it in a simple object.