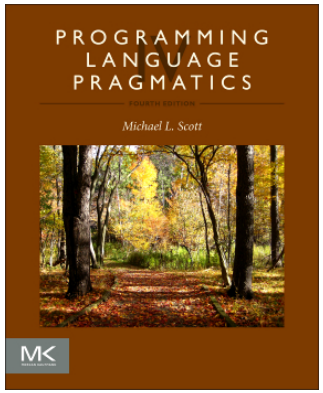
Names, Scopes, and Bindings

17-363/17-663: Programming Language Pragmatics



Reading: PLP chapter 3

ELSEVIER

Name, Scope, and Binding

- Consider this example of a variable binding:

{

S_1

int x = *e*;

S_2

}

- x is a *name*
- **int** x = *e*; is a *binding*
  - associates x with a variable
  - assigns the result of evaluating *e* to the variable
- The *scope* of x is where the binding is active
  - typically the statements S_2 that follow the binding

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

int 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]

var 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

{

S_1

var x;

S_2[x]

x = e[x]

S_3[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
  - 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

```
int* f() {
    int x = 5;
    return &x;
}
int *p = f(); // returns a dangling reference
```

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

- *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

Lifetime and Storage Management

- What does this C code print?

```
    // example C code
int x = 5;
if (y>0) {
    int x = 7;
    print(x);
}
```

Lifetime and Storage Management

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

```
    // example C code
int x = 5;
if (y>0) {
    int x = 7;   // shadows the x=5 binding
    print(x);    // will print 7
}
```

  - 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

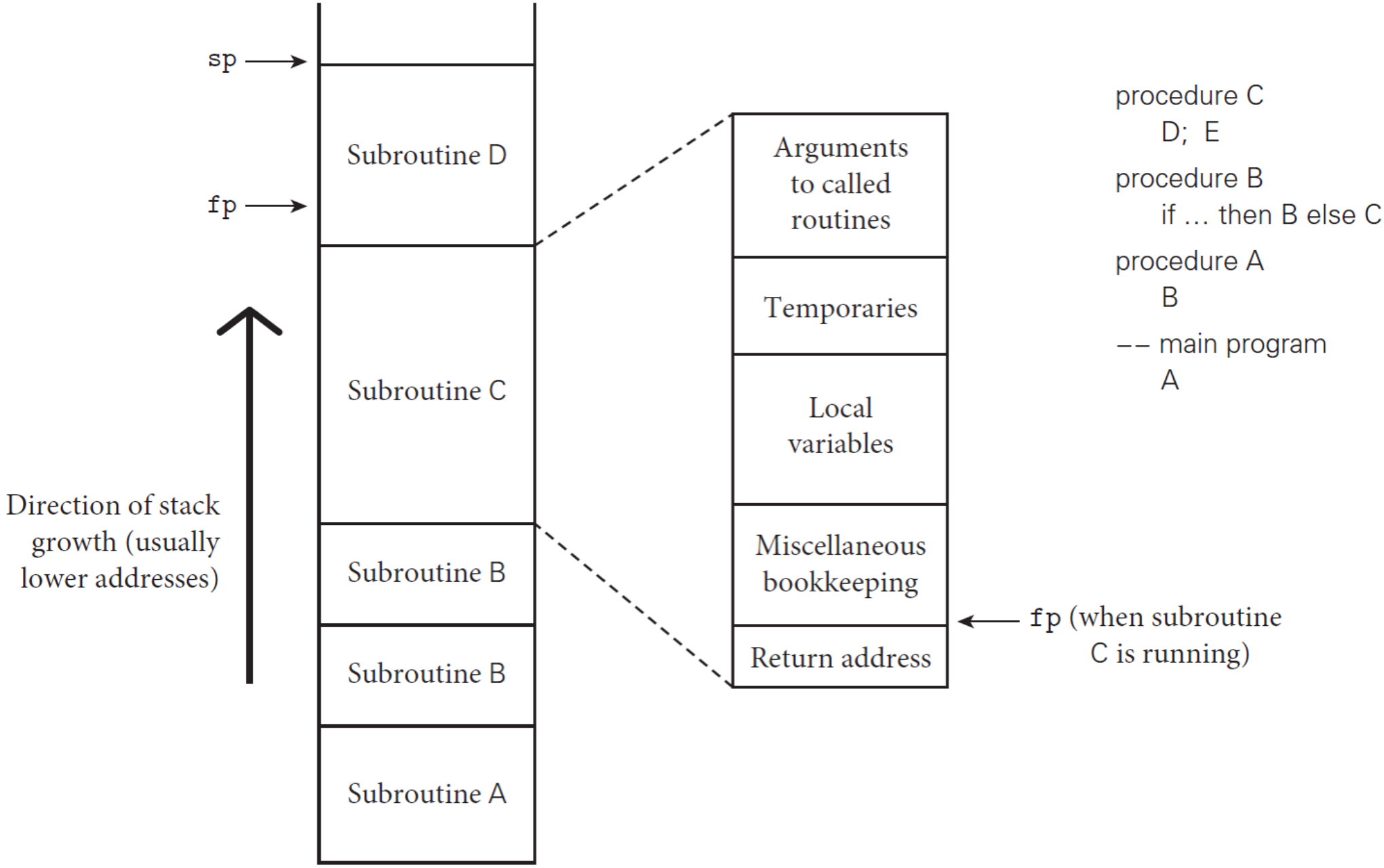
Lifetime and Storage Management

- 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



Lifetime and Storage Management

- 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?

Lifetime and Storage Management

- 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

Lifetime and Storage Management

- Heap for dynamic allocation

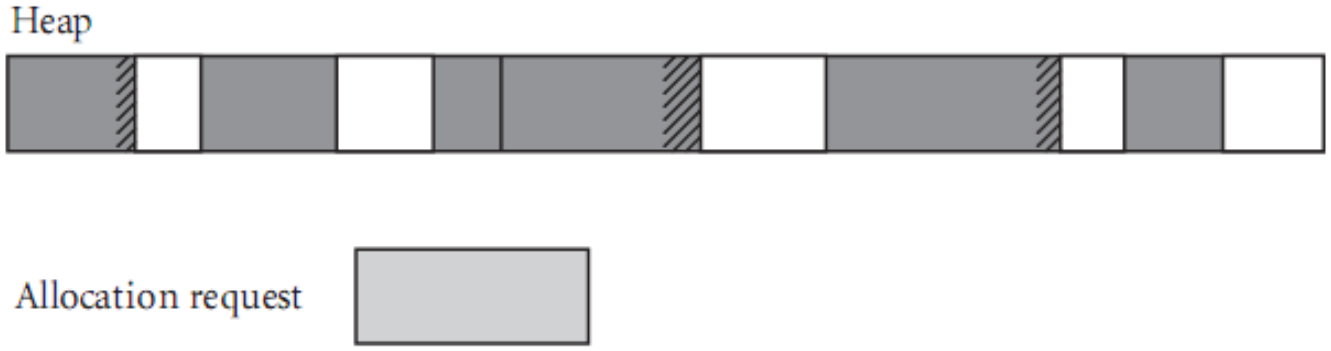


Figure 3.2 Fragmentation. The shaded blocks are in use; the clear blocks are free. Cross-hatched space at the ends of in-use blocks represent internal fragmentation. The discontiguous free blocks indicate external fragmentation. While there is more than enough total free space remaining to satisfy an allocation request of the illustrated size, no single remaining block is large enough.

Declarations and Definitions

- Declarations
  - Introduce a name; give its type (if in a typed language)
- Definitions
  - Fully define an entity
    - Specify value for variables, function body for functions
- Common rules
  - Declaration before use
  - Definition before use
  - Why might we care about these?

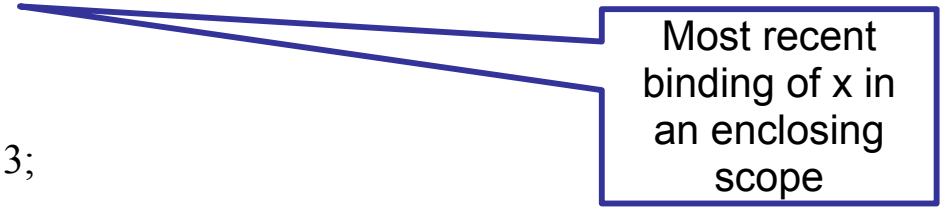
Declarations and Definitions

- Declarations
  - Introduce a name; give its type (if in a typed language)
- Definitions
  - Fully define an entity
    - Specify value for variables, function body for functions
- 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

– 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

Scope Rules

- The classical example of static scope rules is the most closely nested rule used in block structured languages such as Algol 60 and 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

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

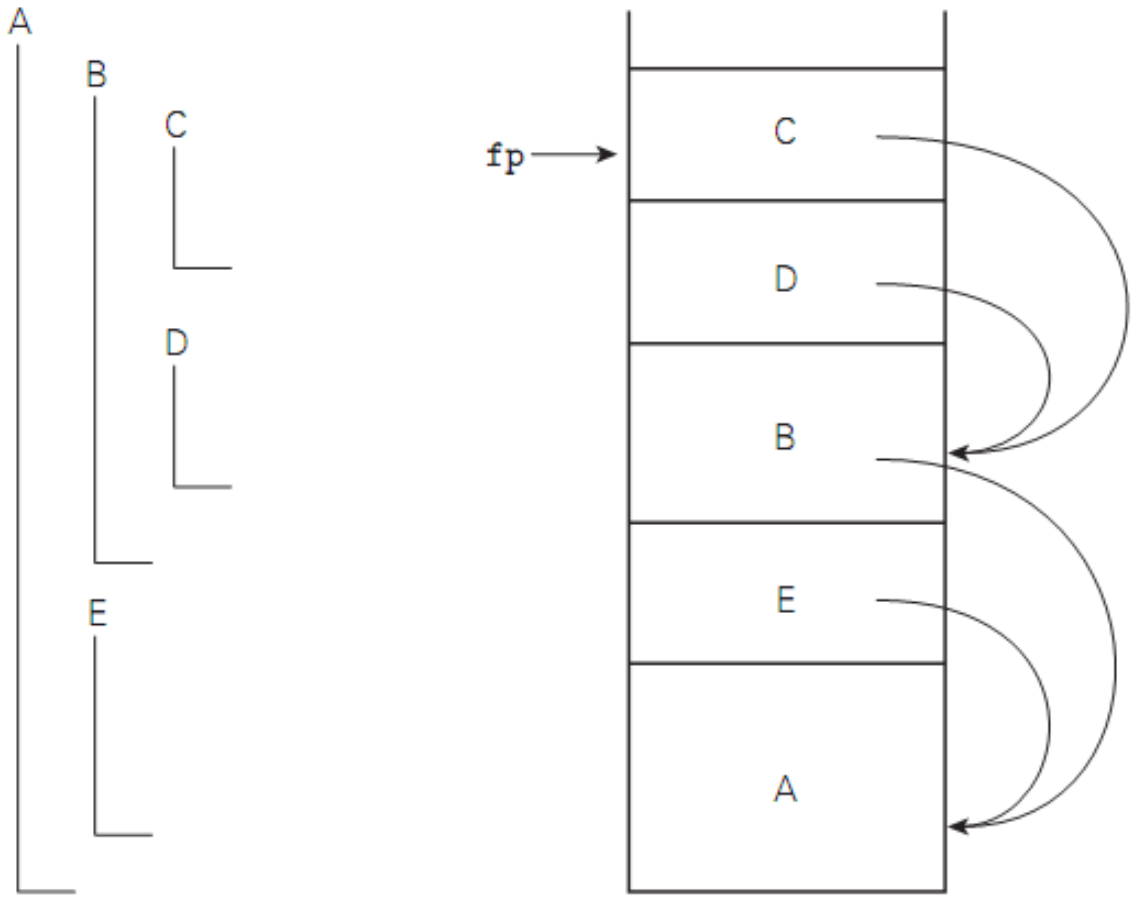


Figure 3.5 **Static chains.** Subroutines A, B, C, D, and E are nested as shown on the left. If the sequence of nested calls at run time is A, E, B, D, and C, then the static links in the stack will look as shown on the right. The code for subroutine C can find local objects at known offsets from the 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.

Dynamic 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

```
function scopes(input, output) {
  var a;
  function first() {
    a = 1;
  }
  function second() {
    var a;
    first();
  }
  a = 2; second(); print(a);
}
```

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

Practice with Scope Rules
Static vs. Dynamic

```
function scopes(input, output) {
  var a;
  function first() {
    a = 1;
  }
  function second() {
    var a;
    first();
  }
  a = 2; second(); print(a);
}
```

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

Dynamic Scope

- Dynamic scope rules are usually encountered in interpreted languages
  - Early LISP dialects used dynamic scope
  - Can be useful for “implicit parameters”
- Languages with dynamic scope don’t usually have static typechecking
  - The compiler can’t determine what variable a name refers to!
- Dynamic scope is now considered a bad design
  - Use static variables or default parameters instead

Binding of Referencing Environments

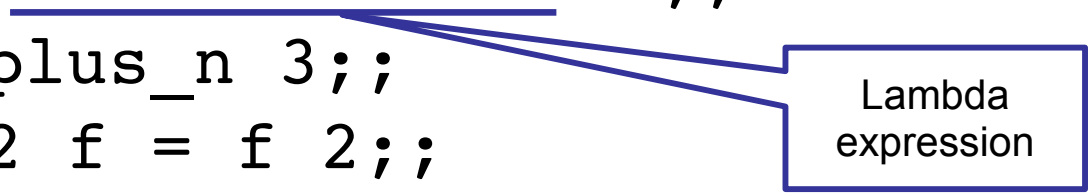
- A *referencing environment* of a statement at run time is the set of active bindings
- A referencing environment corresponds to a collection of scopes that are examined (in order) to find a binding

First Class Functions

- Consider the following OCaml code:

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

apply_to_2 plus3 => 5
```



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

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

Hiding Names with Modules

- Consider the following OCaml code:

```
module Set : sig
  type 'a set
  val make : unit -> 'a set
  val union : 'a set -> 'a set -> 'a set
end = struct
  type 'a set = 'a list
  let make () = ...
  let union_helper(...) = ...
  let union(set1, set2) =
  union_helper(...)
end
```

Signature shows what module clients can see

Signature hides the implementation type for set

Private helper functions are also hidden by leaving them out of the signature

Hiding Names with Modules

- Related facilities in other languages
  - Java: hide elements with **private** keyword
  - C: put public members into header file

- These handle the most common cases, but are not as expressive/elegant as OCaml (or ML) modules
  - Module signatures make public interface explicit
  - Types can be *partially hidden* in a way that's hard to express in other languages

The Meaning of Names within a Scope

Aliasing: when two names refer to the same entity

- Benefits
  - Expressing linked data structures
  - Asymptotically more efficient algorithms
    - e.g. union-find
- Drawbacks
  - Make optimization more difficult
    - FORTRAN prohibits aliasing between procedure arguments
      - Thus for a long time FORTRAN compilers produced faster code than C compilers
    - Now C has a **strict** modifier to do the same thing
  - Confusing to programmers
    - Changing data through one name affects accesses through another
    - Must be used carefully

The Meaning of Names within a Scope

- Overloading: functions with the same name that take arguments of different types
  - Almost every language overloads operators
    - integer + vs. real +
    - choose which one to invoke by types of arguments
    - treat name as if it included the argument types (was literally true in translation from C++ to C)
  - Some languages (e.g. C++) support programmer-defined overloading:

```
int norm (int a) { return a>0 ? a : -a; }
complex norm (complex c ) { // ...
```

Binding Time

- Binding Time is the point at which a binding is created or, more generally, the point at which any implementation decision is made
  - language design time
    - language constructs, built-in types
  - language implementation time
    - I/O, arithmetic overflow, … (if unspecified in manual)

Binding Time

- Implementation decisions (continued):
  - –program writing time
    - •algorithms, names
  - –compile time
    - •data layout
  - –link time
    - •layout of whole program in memory
  - –load time
    - •choice of physical addresses

Binding Time

- Implementation decisions (continued):
  - run time
    - values of variables, sizes of strings and arrays
    - subsumes
      - program start-up time
      - module entry time
      - procedure entry time
      - statement execution time

Binding Time

- The terms *static* and *dynamic* are generally used to refer to things bound before run time and at run time, respectively
  - “static” is a coarse term; so is "dynamic"

Binding Time

- What improves efficiency—early or late binding?
  - Early binding times are associated with greater efficiency
  - Later binding times are associated with greater flexibility
- "Compiled" vs. "interpreted" languages
  - Not a hard distinction—can implement any PL either way
  - Easier to implement compiler when things are bound early (e.g. C)
  - Languages that bind many things late (e.g. Python) are easier to implement with an interpreter

Conclusions

- The morals of the story:
  - language features can be surprisingly subtle
  - designing languages to make life easier for the compiler writer *can* be a GOOD THING
  - most of the languages that are easy to understand are easy to compile, and vice versa

Conclusions

- A language that is easy to compile often leads to
  - a language that is easy to understand
  - more good compilers on more machines (compare Pascal and Ada!)
  - better (faster) code
  - fewer compiler bugs
  - smaller, cheaper, faster compilers
  - better diagnostics