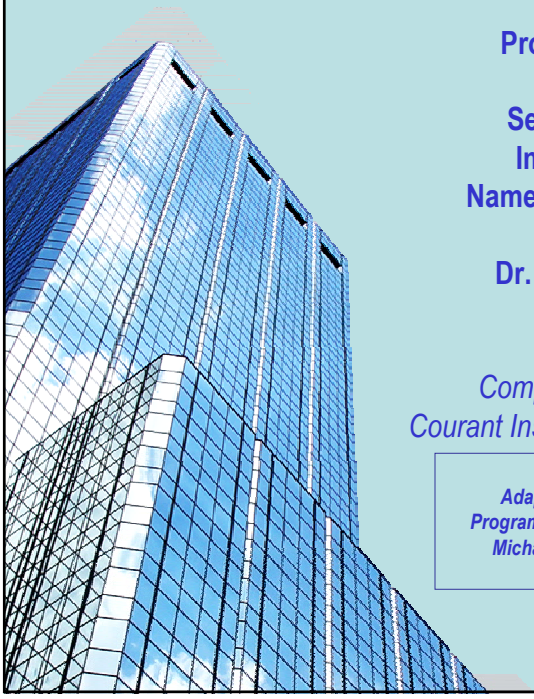


Programming Languages

Session 2 – Main Theme
Imperative Languages:
Names, Scoping, and Bindings

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Adapted from course textbook resources
Programming Language Pragmatics (3rd Edition)
Michael L. Scott, Copyright © 2009 Elsevier

Agenda



1. Session Overview
2. Imperative Languages: Names, Scoping, and Bindings
3. Conclusion

What is the course about?



- Course description and syllabus:
  - » http://www.nyu.edu/classes/jcf/g22.2110-001
  - » http://www.cs.nyu.edu/courses/fall10/G22.2110-001/index.html
- Textbook:
  - » ***Programming Language Pragmatics (3rd Edition)***



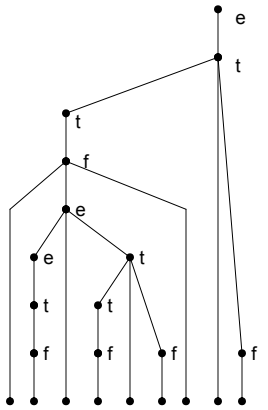
Michael L. Scott
Morgan Kaufmann
ISBN-10: 0-12374-514-4, ISBN-13: 978-0-12374-514-4, (04/06/09)

Session Agenda



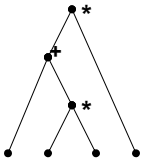
- Session Overview
- Imperative Languages: Names, Scoping, and Bindings
- Conclusion

Review: BNF, Concrete and Abstract Syntax Trees



(A + B * C) * D

```
expr ::= expr "+" term |
         expr "–" term |
         term
term ::= term "*" factor |
         term "/" factor |
         factor
factor ::= number |
           identifier |
           "(" expr ")"
```



A B C D

Icons / Metaphors



Information



Common Realization



Knowledge/Competency Pattern



Governance

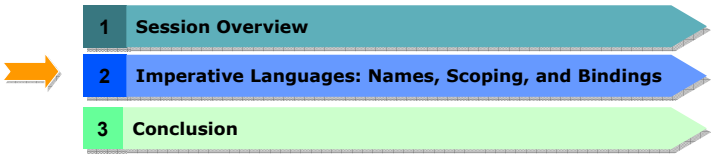


Alignment



Solution Approach

Agenda



Imperative Languages: Names, Scoping, and Bindings - Sub-Topics



- Use of Types
- Name, Scope, and Binding
- Names
- Binding
- Early vs. Late Binding Time Advantages Detailed
- Lifetimes
- Lifetime and Storage Management
- Garbage Collection
- Scopes
- Scope Rules
- Scope Rules – Example: Static vs. Dynamic
- The Meaning of Names within a Scope
- Bindings of Referencing Environments
- Separate Compilation
- Conclusions

Name, Scope, and Binding



- A name is exactly what you think it is
  - » Most names are identifiers
  - » symbols (like '+') can also be names
- A binding is an association between two things, such as a name and the thing it names
- The scope of a binding is the part of the program (textually) in which the binding is active

Use of Types (1/2)

- Operator overloading
- Polymorphism
- Conversion
- Implementation choices
- Error checking
- Differences:
  - » What's typed? The value, the container, or the name
  - » If the name can refer to different containers must they all be the same type?

Use of Types (2/2)

- Integer
- Float
- (In COBOL: Decimal)
- Second-class: “Hollerith”

Names (1/2)



- What can we name?
  - mutable variables
  - values
  - functions
  - types
  - type constructors (e.g., list or vector)
  - classes
  - modules/packages
  - execution points (labels)
  - execution points with environment (continuation)

Names (2/2)



- How do names make programming easier?
  - » Names are more intuitive than numbers when identifying entities.
  - » Names are a key part of abstraction
    - Abstraction reduces conceptual complexity by hiding irrelevant details
    - Names for subroutines: control abstraction
    - Names for classes: data abstraction

Binding (1/6)



- 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
    - program structure, possible type, built-in features such as keywords
  - » language implementation time
    - I/O, arithmetic overflow, type equality (if unspecified in manual)
    - implementation dependent semantics such as bit-width of an integer

Binding (2/6)



- Implementation decisions (continued):
  - » program writing time
    - algorithms, names chosen by programmers
  - » compile time
    - plan for data layout, bindings of high-level constructs to machine code
  - » link time
    - layout of whole program in memory, final bindings of names to addresses
  - » load time
    - choice of physical addresses, Physical addresses (can change during run time)

Binding (3/6)



- Implementation decisions (continued):
  - » run time
    - value/variable bindings, sizes of strings
    - bindings of variables to values, includes many bindings which change during execution
    - subsumes
      - – program start-up time
      - – module entry time
      - – elaboration time (point a which a declaration is first "seen")
      - – procedure entry time
      - – block entry time
      - – statement execution time

Binding (4/6)



- 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"
- *IT IS DIFFICULT TO OVERSTATE THE IMPORTANCE OF BINDING TIMES IN PROGRAMMING LANGUAGES*

Binding (5/6)



- In general, early binding times are associated with greater efficiency
- Later binding times are associated with greater flexibility
- Compiled languages tend to have early binding times
- Interpreted languages tend to have later binding times
- Today we talk about the binding of identifiers to the variables they name

Binding (6/6)



- Scope Rules - control bindings
  - » 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 or Pascal is referenced by pointers, not names

Early vs. Late Binding Time Advantages Detailed



- What are some advantages of early binding times?
  - » Efficiency: the earlier decisions are made, the more optimizations are available to the compiler
  - » Ease of implementation: Earlier binding makes compilation easier
- What are some advantages of late binding times?
  - » Flexibility: Languages that allow postponing binding give more control to programmer
  - » Polymorphic code: Code that can be used on objects of different types is polymorphic, SMALLTALK is an example
- Typically, early binding times are associated with compiled languages and late binding times with interpreted languages.
- But: Even if language has late binding semantics, we often still want to have early binding performance.
- For example, virtual method dispatch in C++ or JAVA has late binding semantics. The implementation typically uses a level of indirection, the so-called vtable. To eliminate this extra memory access for each method invocation, de-virtualization must have knowledge about the entire class hierarchy. In the presence of dynamic linking, that knowledge is only available at link/load time.

Lifetimes (1/2)



- The period of time between the creation and destruction of a name-to-object binding is called the binding's lifetime.
  - The time between the creation of an object and its destruction is the object's lifetime.
- Is it possible for these to be different?
  - When a variable is passed by reference, the binding of the reference variable has a shorter lifetime than that of the object being referenced
  - If there are multiple pointers to an object and one of the pointers is used to delete the object, the object has a shorter lifetime than the bindings of the pointers
  - A pointer to an object that has been destroyed is called a dangling reference and is usually a bug

Lifetimes (2/2)



- For objects residing in memory, there are typically three areas of storage corresponding to different lifetimes:
  - static objects: lifetime of entire program execution
    - globals, static variables in C
  - stack objects: from the time the function or block is entered until the time it is exited
    - local variables
  - heap objects: arbitrary lifetimes, not corresponding to the entrance or exit of a function or block
    - dynamically allocated objects, e.g., with new in C++ or malloc in C

Lifetime and Storage Management (1/12)



- Key events
  - creation of objects
  - creation of bindings
  - references to variables (which use bindings)
  - (temporary) deactivation of bindings
  - reactivation of bindings
  - destruction of bindings
  - destruction of objects

Lifetime and Storage Management (2/12)



- The period of time from creation to destruction is called the LIFETIME of a binding
  - If object outlives binding, it is garbage
  - If binding outlives object, it is a dangling reference
- The textual region of the program in which the binding is *active* is its scope
- In addition to talking about the *scope of a binding*, we sometimes use the word *scope* as a noun all by itself, without an indirect object

Lifetime and Storage Management (3/12)



- Storage Allocation mechanisms
  - Static
  - Stack
  - Heap
- Static allocation for
  - The actual program instructions (in machine code)
  - globals
  - static or own variables
  - explicit constants (including strings, sets, etc)
  - scalars may be stored in the instructions
  - Numeric and string literals
  - Tables produced by the compiler
- Static objects are often allocated in read-only memory so that attempts to change them will be reported as an error by the operating system.
- Under what conditions could local variables be allocated statically?
- The original FORTRAN did not support recursion, allowing local variables to be allocated statically.

Lifetime and Storage Management (4/12)



- Dynamic Allocation
  - For most languages, the amount of memory used by a program cannot be determined at compile time
    - Earlier versions of FORTRAN are exceptions
  - Some features that require dynamic memory allocation:
    - Recursion
    - Pointers, explicit allocation (e.g., **new**)
    - Higher order functions

Lifetime and Storage Management (5/12)



- Central stack 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)
  - » In languages with recursion, the natural way to allocate space for subroutine calls is on the stack.
  - » This is because the lifetimes of objects belonging to subroutines follow a last-in, first-out (LIFO) discipline
  - » Each time a subroutine is called, space on the stack is allocated for the objects needed by the subroutine.
  - » This space is called a stack frame or activation record.

Lifetime and Storage Management (6/12)



- Objects in the activation record/stack frame may contain (cf., Figure 3.1)
  - » arguments and return values
  - » local variables
  - » temporary variables (temporaries)
  - » bookkeeping information (saved registers, line number static link, etc.)
- Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time

Lifetime and Storage Management (7/12)

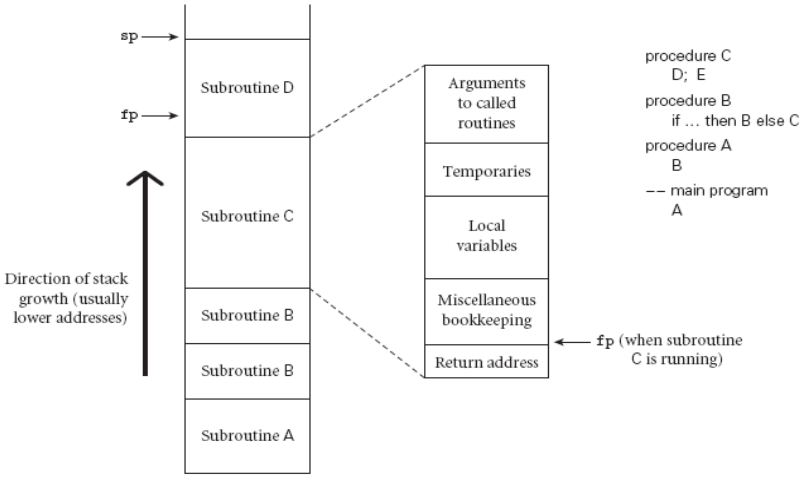


Figure 3.1 Stack-based allocation of space for subroutines. We assume here that subroutines have been called as shown in the upper right. In particular, B has called itself once, recursively, before calling C. If D returns and C calls E, E's frame (activation record) will occupy the same space previously used for D's frame. At any given time, the stack pointer (`sp`) register points to the first unused location on the stack (or the last used location on some machines), and the frame pointer (`fp`) register points to a known location within the frame of the current subroutine. The relative order of fields within a frame may vary from machine to machine and compiler to compiler.

Lifetime and Storage Management (8/12)

- 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

      or
    - combining what's known in both places (interprocedural optimization)

Lifetime and Storage Management (9/12)



- Heap-Based Allocation
  - » Some objects may not follow a LIFO discipline:
    - Space allocated with **new**.
    - Space for local variables and parameters in functional languages
  - » The lifetime of these objects may be longer than the lifetime of the subroutine in which they were created.
  - » These are allocated on the heap: a section of memory set aside for such objects.
  - » (not to be confused with priority queues – a totally different use of the word “heap”)

Lifetime and Storage Management (10/12)



- Heaps
  - » The heap is finite: if we allocate too many objects, we will run out of space.
  - » **Solution**: deallocate space when it is no longer needed by an object.
    - Manual deallocation: with e.g., free, delete (C, PASCAL)
    - Automatic deallocation via garbage collection (JAVA, C#, SCHEME, ML, PERL)
    - Semi-automatic deallocation, using destructors (C++, ADA)
      - – Automatic because the destructor is called at certain points automatically
      - – Manual because the programmer writes the code for the destructor
  - » Manual deallocation is a common source of bugs:
    - Dangling references
    - Memory leaks

Lifetime and Storage Management (11/12)



- Heaps (continued)
  - » The heap starts out as a single block of memory.
  - » As objects are allocated and deallocated, the heap becomes broken into smaller blocks, some in use and some not in use.
  - » Most heap-management algorithms make use of a free list: a singly linked list containing blocks not in use.
    - Allocation: a search is done to find a free block of adequate size
      - – First fit: first available block is taken
      - – Best fit: all blocks are searched to find the one that fits the best
    - Deallocation: the block is put on the free list
  - » Fragmentation is an issue that degrades performance of heaps over time:
    - Internal fragmentation occurs when the block allocated to an object is larger than needed by the object
    - External fragmentation occurs when unused blocks are scattered throughout memory so that there may not be enough memory in any one block to satisfy a request

Lifetime and Storage Management (12/12)



- Heap for dynamic allocation

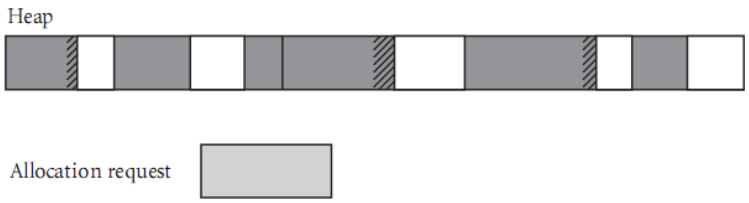


Figure 3.2 **Fragmentation.** The shaded blocks are in use; the clear blocks are free. Cross-hatched space at the ends of in-use blocks represent internal fragmentation. The discontiguous free blocks indicate external fragmentation. While there is more than enough total free space remaining to satisfy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

Garbage Collection (1/9)



- Garbage collection refers to any algorithm for automatic deallocation.
- **Variations**
  - » Mark/sweep
    - Variant: compacting
    - Variant: non-recursive
  - » Copying
    - Variant: incremental
    - Variant: generational

Garbage Collection (2/9)



- During garbage collection, the aim is to deallocate any object that will never be used again.
- How do you know if an object will never be used again?
- An approximation is the set of objects that are live.
- An object x is live if:
  - » x is pointed to by a variable,
    - on the stack (e.g., in an activation record)
    - that is global
  - » there is a register (containing a temporary or intermediate value) that points to x, or
  - » there is another object on the heap (e.g., y) that is live and points to x
- All live objects in the heap can be found by a graph traversal:
  - » Start at the roots: local variables on the stack, global variables, registers
  - » Any object not reachable from the roots is dead and can be reclaimed

Garbage Collection (3/9)



- Mark / Sweep
  - » Each object has an extra bit called the mark bit
  - » Mark phase: the collector traverses the heap and sets the mark bit of each object encountered
  - » Sweep phase: each object whose mark bit is not set goes on the free list

Name	Definition

GC()	**for** each root pointer p **do**
	  mark(p);
	sweep();

mark(p)	**if** p->mark != 1 **then**
	  p->mark := 1;
	  for each pointer field p->x **do**
	  mark(p->x);

sweep()	**for** each object x **in** heap **do**
	  if x.mark = 0 **then** insert(x, free_list);
	  else x.mark := 0;

Garbage Collection (4/9)



- Heap is split into 2 parts: FROM space, and TO space
- Objects allocated in FROM space
- When FROM space is full, garbage collection begins
- During traversal, each encountered object is copied to TO space
- When traversal is done, all live objects are in TO space
- Now we flip the spaces – FROM space becomes TO space and vice-versa
- Since we are moving objects, any pointers to them must be updated: this is done by leaving a forwarding address

Garbage Collection (5/9)



Name	Definition
GC()	**for** each root pointer p **do** p := traverse(p);

traverse(p)	**if** *p contains forwarding address then
	  p := *p; // follow forwarding address
	  return p;
	else {
	  new_p := copy (p, TO_SPACE);
	  *p := new_p; // write forwarding address
	  for each pointer field p->x **do**
	  new_p->x := traverse(p->x);
	  return new_p;
	}

Garbage Collection (6/9)



- Generational Garbage Collection
  - A variant of a copying garbage collector
  - **Observation**: the older an object gets, the longer it is expected to stay around.

    Why?
    - Many objects are very short-lived (e.g., intermediate values)
    - Objects that live for a long time tend to make up central data structures in the program, and will probably be live until the end of the program
  - **Idea**: instead of 2 heaps, use many heaps, one for each "generation"
    - Younger generations collected more frequently than older generations (because younger generations will have more garbage to collect)
    - When a generation is traversed, live objects are copied to the next-older generation
    - When a generation fills up, we garbage collect it

Garbage Collection (7/9)



- Reference Counting
  - **The problem**
    - We have several references to some data on the heap
    - We want to release the memory when there are no more references to it
    - We don't have garbage collection "built-in"
  - **Idea**: Keep track of how many references point to the data, and free it when there are no more.
    - Set reference count to 1 for newly created objects
    - Increment reference count whenever we create a pointer to the object
    - Decrement reference count whenever we a pointer stops pointing to the object
    - When an object's reference count becomes 0, we can free it

Garbage Collection (8/9)



- Reference Counting Example

```
template<typename T> class Ptr {
    T* ptr; size_t* count;
public:
    Ptr(T* ptr = 0) : ptr(ptr), count(new size_t(1)) { }
    Ptr(const Ptr& other) : ptr(other.ptr), count(other.count) {
            ++(*count);
    }
    ~Ptr() { if (0 == --(*count)) { delete ptr; delete count; } }
    Ptr& operator=(const Ptr& right) {
            if (ptr != right.ptr) {
                    delete ptr; delete count;
                    ptr = right.ptr; count = right.count;
                    ++(*count);
            }
            return *this;
    }
    T& operator*() const { return *ptr; }
    T* operator->() const { return ptr; }
    T* raw() const { return ptr; }
    }
```

Garbage Collection (9/9)



- Comparison
  - » What are the advantages and disadvantages of mark/sweep, copying, and reference counting?
    - Garbage Collection: large cost, occurs rarely
    - Reference Counting: small cost occurs frequently
    - GC is faster overall, but RC has more consistent performance
  - » **Costs of garbage collection**

    L = amount of storage occupied by live data

    M = size of heap
    - Mark/sweep: O(L) + O(M) = O(M) sinceM > L
    - Copying: O(L)

      experimental data for LISP: L 0:3 M
    - Mark/Sweep costs more, but is able to use more memory.

Scopes (1/6)



- Scope:
  - » The region of program text where a binding is active is called its scope
  - » Notice that scope is different from lifetime
- Two major scoping disciplines:
  - » Static or lexical: binding of a name is given by its declaration in the innermost enclosing block
    - binding of a name is determined by rules that refer only to the program text
    - Typically, the scope is the smallest block in which the variable is declared
    - Most languages use some variant of this
    - Scope can be determined at compile timeMost languages use some variant of this
  - » dynamic: binding of a name is given by the most recent declaration encountered at runtime
    - binding of a name is given by the most recent declaration encountered during run-time
    - Used in SNOBOL, APL, some versions of LISP

Scopes (2/6)



- Scoping Example:

```
var x = 1;
function f () { print x; }
function g () { var x = 10; f(); }
function h () { var x = 100; f(); }
f(); g(); h();
```

Scoping	Output
Static	1 1 1
Dynamic	1 10 100

Scopes (3/6)



Static Scoping: Declaration Order

What is the scope of x?

```
{
    statements1 ;
    var x = 5;
    statements2 ;
}
```

- C++, Ada: statements2
- Javascript: entire block
- Pascal: entire block, but not allowed to be used in statements1!

Scopes (4/6)



Declarations and Definitions

- C and C++ require names to be declared before they are used.
- This requires a special mechanism for recursive data types.
- C and C++ solve the problem by separating declaration from definition.
  - » A declaration introduces a name and indicates the scope of the name
  - » A definition describes the object to which the name is bound

Scopes (5/6)



- Declarations and Definitions – Example:

```
class Manager; // Declaration
class Employee {
    Manager* boss;
    Employee* next;
    ...
};
class Manager { // Definition
    Employee* firstEmployee;
    ...
};
```

Scope (6/6)



- Redeclarations
  - » Interpreted languages often allow redeclaration: creating a new binding for a name that was already given a binding in the same scope

```
function addx(int x) { return x + 1; }
function add2(int x)
{
   x := addx(x);
   x := addx(x);
   return x;
}
function addx(int x) { return x + x; }
```

  - » What happens if we call add2(2) ?
    - In most languages, the new definition replaces the old one in all contexts, so we would get 8.
    - In ML, the new meaning only applies to later uses of the name, not previous uses. ML would give 4.

Scope Rules (1/12)



- A *scope* is a program section of maximal size in which no bindings change, or at least in which no re-declarations are permitted (see below)
- In most languages with subroutines, we OPEN a new scope on subroutine entry:
  - » create bindings for new local variables,
  - » deactivate bindings for global variables that are re-declared (these variable are said to have a "hole" in their scope)
  - » make references to variables

Scope Rules (2/12)



- On subroutine exit:
  - » destroy bindings for local variables
  - » reactivate bindings for global variables that were deactivated
- Algol 68:
  - » ELABORATION = process of creating bindings when entering a scope
- Ada (re-popularized the term elaboration):
  - » storage may be allocated, tasks started, even exceptions propagated as a result of the elaboration of declarations

Scope Rules (3/12)



- With STATIC (LEXICAL) SCOPE RULES, a scope is defined in terms of the physical (lexical) structure of the program
  - » The determination of scopes can be made by the compiler
  - » All bindings for identifiers can be resolved by examining the program
  - » Typically, we choose the most recent, active binding made at compile time
  - » Most compiled languages, C and Pascal included, employ static scope rules

Scope Rules (4/12)



- Static Scoping: Nested Scope
  - » Some languages allow nested subroutines or other kinds of nested scopes
  - » Typically, bindings created inside a nested scope are not available outside that scope.
  - » On the other hand bindings at one scope typically are available inside nested scopes. The exception is if a new binding is created for a name in the nested scope.
  - » In this case, we say that the original binding is hidden, and has a hole in its scope.
  - » Some languages allow nested scopes to access hidden binding by using a qualifier or scope resolution operator .
    - In ADA, A:X refers to the binding of X created in subroutine A, even if there is a lexically closer scope
    - In C++, A :: X refers to the binding of X in class A, and :: X refers to the global scope

Scope Rules (5/12)



- 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

Scope Rules (6/12)



- Static Scoping: Nested Subroutines
  - » How does a nested subroutine find the right binding for an object in a outer scope?
    - Maintain a static link to the “parent frame”
    - The parent frame is the most recent invocation of the lexically surrounding subroutine
    - The sequence of static links from the current stack frame to the frame corresponding to the outermost scope is called a static chain
  - » **Finding the right binding**
    - The level of nesting can be determined at compile time
    - If the level of nesting is j, the compiler generates code to traverse the static chain j times to find the right stack frame

Scope Rules (7/12)



- We will see classes - a relative of modules - later on, when discussing abstraction and object-oriented languages
  - » These have even more sophisticated (static) scope rules
- Euclid is an example of a language with lexically-nested scopes in which all scopes are closed
  - » rules were designed to avoid ALIASES, which complicate optimization and correctness arguments

Scope Rules (8/12)



- Note that the bindings created in a subroutine are destroyed at subroutine exit
  - The modules of Modula, Ada, etc., give you closed scopes without the limited lifetime
  - Bindings to variables declared in a module are inactive outside the module, not destroyed
  - The same sort of effect can be achieved in many languages with *own* (Algol term) or *static* (C term) variables (see Figure 3.5)

Scope Rules (9/12)



- Access to non-local variables STATIC LINKS
  - Each frame points to the frame of the (correct instance of) the routine inside which it was declared
  - In the absence of formal subroutines, *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

Scope Rules (10/12)

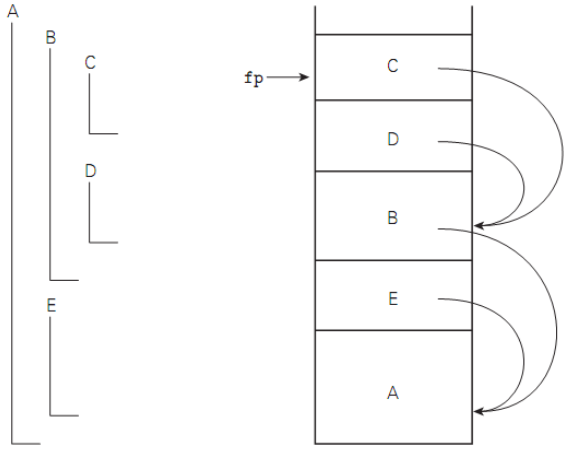


Figure 3.5 **Static chains.** Subroutines A, B, C, D, and E are nested as shown on the left. If the sequence of nested calls at run time is A, E, B, D, and C, then the static links in the stack will look as shown on the right. The code for subroutine C can find local objects at known offsets from the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

Scope Rules (11/12)

- The key idea in **static scope rules** is that bindings are defined by the physical (lexical) structure of the program.
- With **dynamic scope rules**, bindings depend on the current state of program execution
  - » They cannot always be resolved by examining the program because they are dependent on calling sequences
  - » To resolve a reference, we use the most recent, active binding made at run time

Scope Rules (12/12)



- Dynamic scope rules are usually encountered in interpreted languages
  - » early LISP dialects assumed dynamic scope rules.
- Such languages do not normally have type checking at compile time because type determination isn't always possible when dynamic scope rules are in effect

Scope Rules – Example: Static vs. Dynamic (1/4)



```
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; write(a);
end.
```

Scope Rules – Example: Static vs. Dynamic (2/4)



- If static scope rules are in effect (as would be the case in Pascal), the program prints a `1`
- If dynamic scope rules are in effect, the program prints a `2`
- Why the difference? At issue is whether the assignment to the variable `a` in procedure *first* changes the variable `a` declared in the main program or the variable `a` declared in procedure *second*

Scope Rules – Example: Static vs. Dynamic (3/4)



- Static scope rules require that the reference resolve to the most recent, compile-time binding, namely the global variable `a`
- Dynamic scope rules, on the other hand, require that we choose the most recent, active binding at run time
  - » Perhaps the most common use of dynamic scope rules is to provide implicit parameters to subroutines
  - » This is generally considered bad programming practice nowadays
    - Alternative mechanisms exist
      - – static variables that can be modified by auxiliary routines
      - – default and optional parameters

Scope Rules – Example: Static vs. Dynamic (4/4)



- At run time we create a binding for `a` when we enter the main program.
- Then we create another binding for `a` when we enter procedure *second*
  - » This is the most recent, active binding when procedure *first* is executed
  - » Thus, we modify the variable local to procedure *second*, not the global variable
  - » However, we write the global variable because the variable `a` local to procedure second is no longer active

The Meaning of Names within a Scope (1/4)



- Aliasing
  - » What are aliases good for? (consider uses of FORTRAN equivalence)
    - space saving - modern data allocation methods are better
    - multiple representations - unions are better
    - linked data structures - legit
  - » Also, aliases arise in parameter passing as an unfortunate side effect
    - Euclid scope rules are designed to prevent this

The Meaning of Names within a Scope (2/4)



- Overloading
  - » some overloading happens in almost all languages
    - integer + v. real +
    - read and write in Pascal
    - function return in Pascal
  - » some languages get into overloading in a big way
    - Ada
    - C++

The Meaning of Names within a Scope (3/4)



- It's worth distinguishing between some closely related concepts
  - » overloaded functions - two different things with the same name; in C++
    - overload norm

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

  - » polymorphic functions -- one thing that works in more then one way
    - in Modula-2: function min (A : array of integer); …
    - in Smalltalk

The Meaning of Names within a Scope (4/4)



- It's worth distinguishing between some closely related concepts (2)
  - » generic functions (modules, etc.) - a syntactic template that can be instantiated in more than one way *at compile time*
    - via macro processors in C++
    - built-in in C++
    - in Clu
    - in Ada

Binding of Referencing Environments (1/5)



- Accessing variables with dynamic scope:
  - » (1) keep a stack (*association list*) of all active variables
    - When you need to find a variable, hunt down from top of stack
    - This is equivalent to searching the activation records on the dynamic chain

Binding of Referencing Environments (2/5)



- Accessing variables with dynamic scope:
  - (2) keep a central table with one slot for every variable name
    - If names cannot be created at run time, the table layout (and the location of every slot) can be fixed at compile time
    - Otherwise, you'll need a hash function or something to do lookup
    - Every subroutine changes the table entries for its locals at entry and exit.

Binding of Referencing Environments (3/5)



- (1) gives you slow access but fast calls
- (2) gives you slow calls but fast access
- In effect, variable lookup in a dynamically-scoped language corresponds to symbol table lookup in a statically-scoped language
- Because static scope rules tend to be more complicated, however, the data structure and lookup algorithm also have to be more complicated

Binding of Referencing Environments (4/5)



- 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

Binding of Referencing Environments (5/5)



- SCOPE RULES determine that collection and its order
- BINDING RULES determine which instance of a scope should be used to resolve references when calling a procedure that was passed as a parameter
  - » they govern the binding of referencing environments to formal procedures

Separate Compilation (1/3)



- Separately-compiled files in C provide a sort of *poor person's modules*:
  - » Rules for how variables work with separate compilation are messy
  - » Language has been jerry-rigged to match the behavior of the linker
  - » *Static* on a function or variable *outside* a function means it is usable only in the current source file
    - This *static* is a different notion from the *static* variables inside a function

Separate Compilation (2/3)



- Separately-compiled files in C (continued)
  - » *Extern* on a variable or function means that it is declared in another source file
  - » Functions headers without bodies are *extern* by default
  - » *Extern* declarations are interpreted as forward declarations if a later declaration overrides them

Separate Compilation (3/3)



- Separately-compiled files in C (continued)
  - » Variables or functions (with bodies) that don't say *static* **or** *extern* are either *global* or *common* (a Fortran term)
    - Functions and variables that are given initial values are *global*
    - Variables that are not given initial values are *common*
  - » Matching common declarations in different files refer to the same variable
    - They also refer to the same variable as a matching *global* declaration

Conclusions (1/2)



- 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 (2/2)



- 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

Appendix: Fortran Example

C ← FOR COMMENT / STATEMENT NUMBER	CONTINUATION	FORTRAN STATEMENT	IDENTIFICATION
C		PROGRAM FOR FINDING THE LARGEST VALUE	
C	X	ATTAINED BY A SET OF NUMBERS	
		DIMENSION A(999)	
		FREQUENCY 30(2,1,10), 5(100)	
		READ 1, N,.(A(I), I = 1,N)	
1		FORMAT (I3/(12F6.2))	
		BIGA = A(1)	
5		DO 20 I = 2,N	
30		IF (BIGA-A(I)) 10,20,20	
10		BIGA = A(I)	
20		CONTINUE	
		PRINT 2, N, BIGA	
2		FORMAT (22H1THE LARGEST OF THESE I3, 12H NUMBERS IS F7.2)	
		STOP 77777	

Appendix: Fortran - Name Spaces

- Global
  - » Procedures and Functions
  - » Common Blocks
- Statically Scoped
  - » Variables
- Name Spaces
  - » One global name space
  - » One per procedure/function
- Dynamically Scoped
  - » None

Appendix: Fortran - Static and Runtime Naming; Scope

```
PROGRAM MAIN
…
COMMON/ FOO/A,B,I/  BAR/C, D/  BAZ/E, F,
     J
DIMENSION Z(999)
…
CALL SWAP(W, Z(200))
…
SUBROUTINE SWAP(X,Y)
COMMON/ FOO/OLDX /GORP/K
TEMPX = X
X = Y
Y = TEMPX
OLDX = TEMPX
…
FUNCTION N (I)
COMMON/ BAZ/X/GORP/L
NTEMP = I**2 – 3*I + 2
N = NTEMP * L
…
```

Appendix: References - Invisible but Significant

- EQUIVALENCE
- COMMON
- Binding

Appendix - Control flow

- “Spaghetti” code
- GOTO and assigned GOTO
- Can jump anywhere except:
  - » To another procedure
  - » Inside a DO loop from outside the loop
- IF tests and jumps, not blocks
- DO loops and jumps

Appendix - Everything static

- Dimensions of arrays (although not necessarily visible to subroutines)
- Number of instances of storage blocks
- All programs both static and global (except statement functions)
- All I/O devices static and global

Appendix - What can go wrong

- Syntax problems
  - » DO I = 1.3
- Passing a constant
  - » CALL SWAP(2, 3)
- Unchecked Bounds
- Uninitialized Variables

Appendix: Algol 60 - Static and Runtime Naming; Scope

```
PROCEDURE MAIN;
…
x := read();
BEGIN
  INTEGER ARRAY FOO[1:X];
  …
   j := 20;
   blat(j, FOO(j));
  …
PROCEDURE blat(x,y);
 BEGIN
  x := 1000; y:= 1000
 END
…
INTEGER PROCEDURE random;
  BEGIN
   OWN INTEGER seed;
   random := seed := some new value …
  END
…
```


Appendix: New features in Algol 60

- Call by name
- Dynamic array bounds
- Recursive creation of local stack variables
- Own variables
- Nested declarations, scopes
- Inner procedures

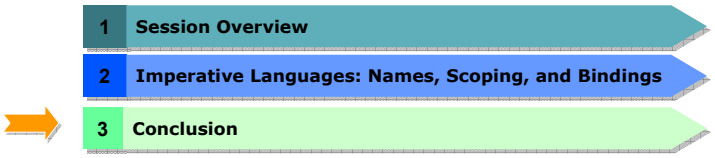
Appendix: Beyond - Algol 68; PL/I; C

- Fully dynamic storage – e.g. PL/I's storage classes:
  - » STATIC – like FORTRAN (local & external)
  - » AUTOMATIC – like Algol 60 non-own
  - » CONTROLLED – dynamically allocated
  - » X BASED(Y) – dynamically allocated with pointer

Appendix: New things that can go wrong

- Unbounded memory
- Dangling references via pointers
- Dangling closures

Agenda



Assignments & Readings



- Readings
  - Chapter 3 (Sections 3.1-3.4 in particular)
- Assignment #2
  - See Assignment #2 posted under “handouts” on the course Web site
  - Due on February 24, 2011

Next Session: Control Structures–Loops, Conditionals, and Case Stmts

