CS 326
Programming Languages, Concepts and Implementation

Instructor: Mircea Nicolescu

Names, Scopes, and Bindings

Language Specification

- General issues in the design and implementation of a language:
  - Syntax and semantics
  - Naming, scopes and bindings
  - Control flow
  - Data types
  - Subroutines

- Issues specific to particular classes of languages:
  - Data abstraction and object orientation
  - Non-imperative programming models (functional and logic languages)
  - Concurrency

Names, Bindings and Scopes

- A name is exactly what you think it is
  - Textbook version: "a name is a mnemonic character string used to represent something else"
  - Most names are identifiers (alpha-numeric tokens), though symbols (like '+') can also be names

- A binding is an association between two entities, such as a name and the entity it names

- The scope of a binding is the part of the program (textually) in which the binding is active

Naming Issues

- Enforced by the language specification:
  - how long can a name be?
  - what characters can be used?
    - @#$%^& is a legal name in Scheme but not in C
  - are names case sensitive?
    - C - yes
    - Pascal - no
    - Prolog - more complex, variables must start with uppercase letters, constants with lowercase letters

- Not enforced, but recommended as good programming practices:
  - C under Windows ("Hungarian notation"): szName, bBooleanVar, fFloatVar, hwndWindow
  - C++ with MFC: m_iIntVar, OnMouseClick()

Binding Time

- The binding time is the point in time at which a binding is created or, more generally, the point at which any implementation decision is made.

- Examples:
  - language design time
    - control flow constructs - if, while...
    - fundamental (primitive) types - int, float
  - language implementation time
    - coupling of I/O operations to the operating system's file implementation
    - handling of run-time exceptions - arithmetic overflow
    - precision (number of bits) for primitive types
  - program writing time
    - algorithms, choosing names

Binding Time

- Examples (cont.):
  - compile time
    - mapping of high-level constructs to machine code
    - layout of (static) data in the memory
  - link time
    - layout of whole program in memory
    - bindings between names and objects in different modules
  - load time
    - conversion from virtual to physical addresses

Binding Time

- Examples (cont.):
  - run time
    - bindings of values to variables
    - includes
      - program start-up time
      - module entry time
      - elaboration time (point a which a declaration is first "seen")
      - procedure call time
      - block entry time
      - statement execution time

Binding Time

- Static vs. Dynamic:
  - static binding time - corresponds to bindings made before run time
  - dynamic binding time - corresponds to bindings made at run time
- Clearly static binding time is a coarse term that can mean many different times (language design, program writing, compilation, etc)
  - also called early binding
- Dynamic is also a coarse term, generally referring to binding times such as when variable values are bound to variables
  - also called late binding

Binding Time

- Early binding
  - associated with greater efficiency
  - compiled languages tend to have early binding times
  - the compiler analyzes the syntax and semantics of global variable declarations only once, decides on a data layout in memory, generates efficient code to access them

- Late binding
  - associated with greater flexibility
  - interpreted languages tend to have late binding times
  - the interpretor analyzes the declarations every time the program runs
  - bindings are not "frozen" at compile time, they can change during execution

Object Lifetime and Binding Lifetime

- Distinguish between names and objects they refer
- Identify several key events:
  - creation of objects
    - allocation
  - creation of bindings
    - declaration
  - references to names (variables, subroutines, types)
    - use of variable in expression, function call
  - temporary deactivation / reactivation of bindings
    - entering a procedure / returning from a procedure (for global variables hidden by local ones)
  - destruction of bindings
    - returning from a procedure (for local variables), end of program (globals)
  - destruction of objects
    - deallocation

Object Lifetime and Binding Lifetime

- Lifetime - the time interval between creation and destruction
- Both objects and bindings have their own, possibly distinct lifetimes

- If an object outlives its (only) binding it's garbage

```
p = new int ;
p = NULL ;
```

- If a binding outlives its object it's a dangling reference

```
p = new int ;
r = p ;
delete r ;
```

Storage Management

- Lifetime of objects is determined by allocation and deallocation

- Allocation - getting ("reserving") a memory cell from some pool of available cells (the "free space")
- Deallocation - placing a memory cell back in the pool

- Storage allocation mechanisms:
  - Static
  - Dynamic
    - stack
    - heap
      - explicit
      - implicit

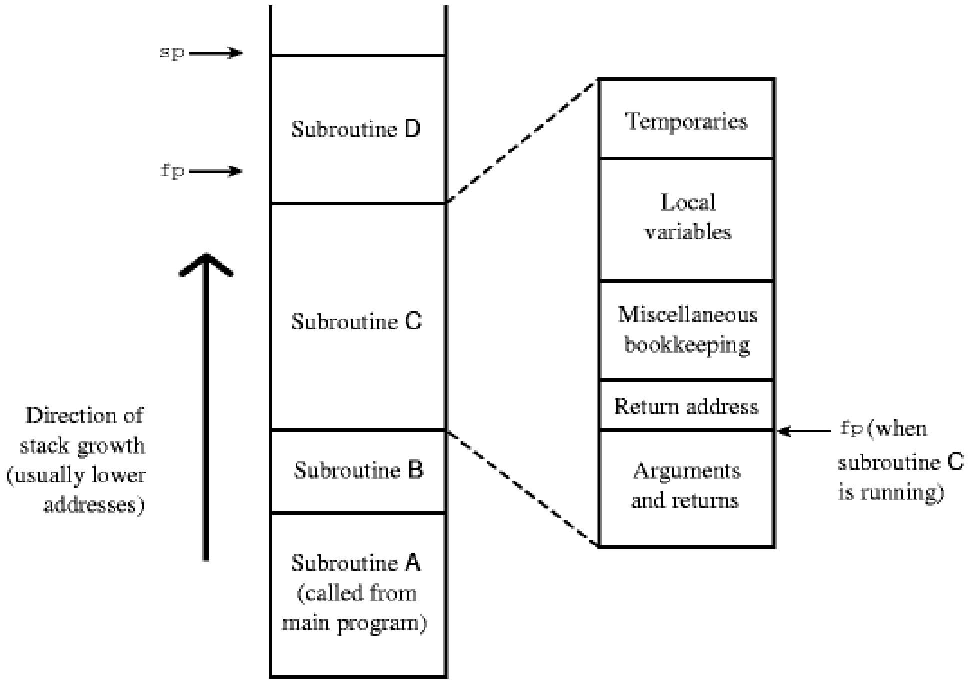
Static Allocation

- Static allocation
  - Each object is given an address before execution begins and retains that address throughout execution
  - Examples - program code, C global variables and static variables, all Fortran 77 variables, explicit constants (including character strings), tables for debugging

Dynamic Stack Allocation

- Stack-based allocation
  - Objects are allocated (on a stack), and bindings are made when entering a subroutine
  - They are destroyed when returning from subroutine
  - Corresponds to last-in, first-out order
  - Examples - arguments, local variables, return value, return address, temporaries

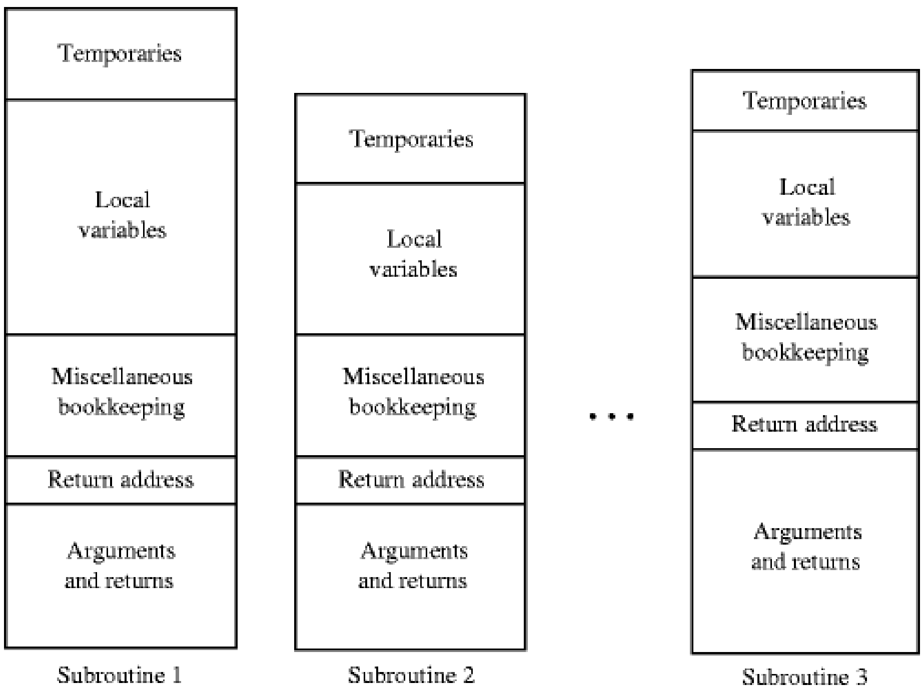
Dynamic Stack Allocation



- Frame (a.k.a. activation record) - an entry on the stack
  - when a subroutine is invoked - push a frame on the stack
  - when a subroutine ends - pop a frame from the stack
- Stack pointer (sp) - register that points to the first unused entry on the stack
- Frame pointer (fp) - register that points to a known location within the active frame
  - Objects within frame can be accessed at a predefined offset from fp

Special Case

- Static allocation for local items in subroutines (Fortran 77):



- Advantages:
  - efficiency (avoid stack maintenance)
  - efficiency (direct addressing)
  - history-sensitive local variables
- Disadvantages:
  - inefficiency (all local items stay allocated all the time
  - lack of flexibility (no recursion!)

Dynamic Stack Allocation

- Stack-based allocation – cont.
- Advantages:
  - Allows recursion
  - Reuses space
- Disadvantages:
  - Run-time overhead of allocation and deallocation on stack
  - Local variables cannot be history sensitive
  - Inefficient references (indirect addressing)

Dynamic Stack Allocation

- Maintenance of the stack is the responsibility of:
  - calling sequence - code executed by caller immediately before and after the call
  - subroutine prologue and epilogue - code executed by subroutine at its beginning / end

- Which is more efficient?

```
#define max(x,y) x>y?x:y          OR          int max (int x, int y)
                                              { return x>y?x:y ; }
```

  - The macro (#define) is generally more efficient, as it does not have the overhead of stack manipulation

Dynamic Heap Allocation

- Explicit heap-based allocation
  - Allocated and deallocated by explicit directives at arbitrary times, specified by the programmer
  - Take effect during execution
  - Examples - dynamic objects in C (via malloc and free), or C++ (via new and delete)

Dynamic Heap Allocation

- Implicit heap-based allocation
  - Allocation and deallocation are implicit (transparent for the programmer)
  - Example:
    - allocation of list structures in Scheme: (define x (cons 'a '(b c)))

- Advantage:
  - Flexibility, ease of use

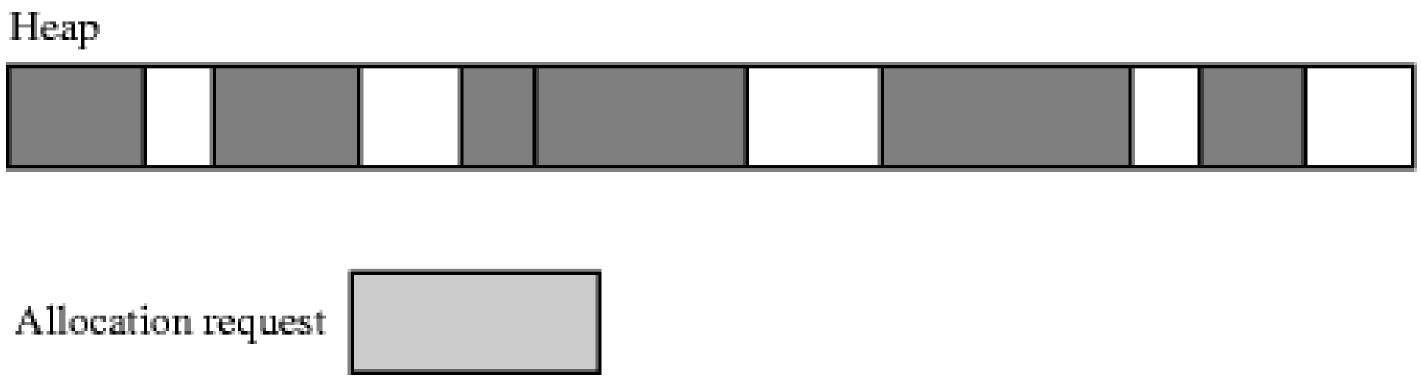
- Disadvantage:
  - Possible inefficiency
    - if the programmer knows that there will be N elements in a list, it would be better to explicitly allocate space for them all at once

Dynamic Heap Allocation

- Allocation is made in a memory region called heap - no connection with the heap data structure
- Principal concerns in heap management are speed and space
- Space issues:
  - Internal fragmentation
    - when allocating a block larger than required to hold a given object
    - the extra space in the block is unused
  - External fragmentation
    - when allocated blocks are scattered through the heap, making the free space extremely fragmented
    - there may be a lot of free space, but no piece is large enough for some future request

Dynamic Heap Allocation

- External fragmentation:



  - Shaded blocks – in use
  - Clear blocks – free

Dynamic Heap Allocation

- Dealing with external fragmentation
  - cannot totally avoid it
  - ability of the heap to satisfy requests may degrade over time

- The solution:
  - compact the heap by moving already allocated blocks

- Why is this difficult?
  - need to find all pointers that refer to the moved blocks, and update their values

Dynamic Heap Allocation

- Implementation
  - Maintain a single linked list of heap blocks that are not currently used (the free list)

- Strategies:
  - First fit – select the first block in the list that is large enough to satisfy the allocation request
  - Best fit – select the smallest block in the list that is large enough to satisfy the allocation request

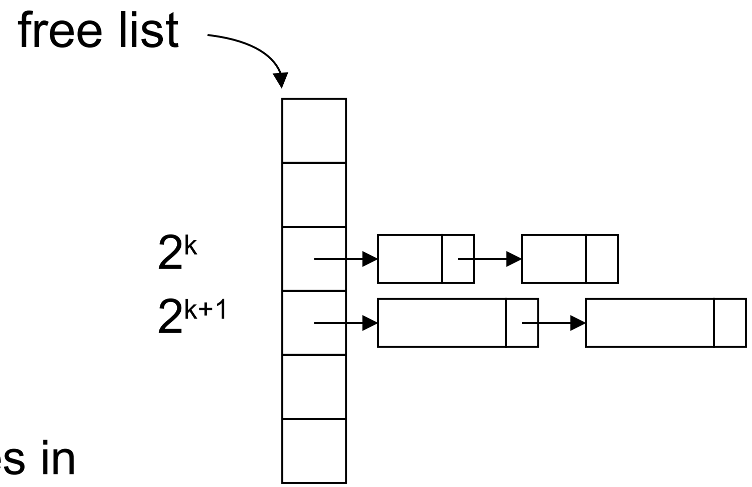
- First fit
  - Faster, tends to produce internal fragmentation
- Best fit
  - Slower (searches the entire list), less internal fragmentation

Dynamic Heap Allocation

- Using a single linked list makes the allocation time linear in the number of free blocks
- To reduce it to constant time:
- Implementation:
  - separate lists for blocks of different sizes
- Strategies:
  - Buddy system
  - Fibonacci heap

Dynamic Heap Allocation

- Buddy system:



  - Block sizes are powers of 2
  - Allocation:
    - a request for a block of size 2^k comes in
    - if a block of size 2^k is available, take it
    - if not, split a block of size 2^{k+1} in two halves (2^k each), use half for allocation, and place the other in the 2^k free list
  - Deallocation:
    - merge the block with its "buddy" (the other half) if it is free

- Fibonacci system – similar, but uses Fibonacci numbers instead of powers of 2

Announcements

- Readings
  - Rest of Chapter 3
- Homework
  - HW 3 out – due on February 29
  - Submission
    - at the beginning of class
    - with a title page: Name, Class, Assignment #, Date
    - preferably typed

Garbage Collection

- If heap-based allocation is explicit (such as in C), responsibility of deallocation (free, delete) stays with the programmer
  - Advantages: implementation simplicity, speed
  - Disadvantages: burden on programmer, manual deallocation errors are among most common bugs, and also most difficult to detect
- If heap-based allocation is implicit (such as in Scheme), deallocation must be also implicit
  - Need to check if an object is not referenced by any variable before deallocating it
  - Must provide a garbage colection mechanism to reclaim "unreachable" objects
  - Advantages: convenience, safety
  - Disadvantages: complexity in implementation, run-time overhead

Scope Rules

- Lifetime of a binding - the period of time from creation to destruction of the binding
- Scope of a binding - the textual region of the program in which the binding is active

- Examples of scopes:
  - the "global" scope (the entire program) - for global variables
  - "local" scopes (subroutines, blocks between { } in C++) - for local variables

Scope Rules

- In most languages with subroutines:
  - open a new scope on subroutine entry
  - create bindings for new local variables (process also called *elaboration*)
  - deactivate bindings for global variables that are hidden by local ones with same name (these global variable are said to have a "hole" in their scope)
  - on subroutine exit, destroy bindings for local variables and reactivate bindings for global variables that were deactivated (hidden)

Scope Rules

- Languages can be statically or dynamically scoped

- Statically (also called lexically) scoped
  - The scope for a binding can be determined by examining the program text
  - Scopes are determined at compile time
  - Examples: C, Pascal

- Dynamically scoped
  - Scopes depend on the flow of control at run time
  - Scopes cannot be determined by examining the program (at compile time), because they depend on (dynamic) calling sequences
  - Examples: APL, Snobol, early Lisp

Scope Rules

- Referencing environment
  - represents the set of active bindings at a given point in program execution
  - determined by static or dynamic scope rules
  - corresponds to a sequence of scopes that can be examined (in order) to find the current binding for a given name
- Binding rules
  - can be deep binding or shallow binding
  - they also determine the referencing environment
  - assume a function is passed as argument and later called (Scheme)
  - when the function is called, what referencing environment will it use?
    - deep binding - use the environment from the moment when function is passed as argument
    - shallow binding - use the environment from the moment of function call

Static Scope

- In a language with static scoping, scopes can be fully determined at compile time, by examining the program text
- Most compiled languages, C and Pascal included, employ static scope rules
- The simplest case – the current binding for a name is the one encountered most recently in a top-to-bottom scan of the program (in early Basic – only a single, global scope)
- How to deal with nested scopes?

Static Scope

- Nested scopes
  - Typically introduced by definitions of subroutines inside each other (in Algol, Pascal, Ada)
- Closest nested scope rule:
  - A name introduced in a declaration is known:
    - in the scope where it's declared, and
    - in each internally nested scope, unless it's hidden by another declaration of the same name
- To find the object referenced by a name:
  - Look for a declaration with that name in the current scope
  - If there is one, that defines the binding
  - If not, look in the immediate surrounding (outer) scope
  - Continue looking outward until a declaration is found for that name
  - If the outermost (global) scope is reached without success → error

Static Scope

- Structure of a Pascal procedure:

```
procedure P (<name> : <type>, <name> : <type>, ...);
<declarations of variables and subroutines>
begin
   <body of P>
end;
```

- A function is similar, it only needs to return something:

```
function F (<name> : <type>, <name> : <type>, ...) : <type>;
<declarations of variables and subroutines>
begin
   <body of F>
end;
```

Static Scope

- Nested scopes - example:

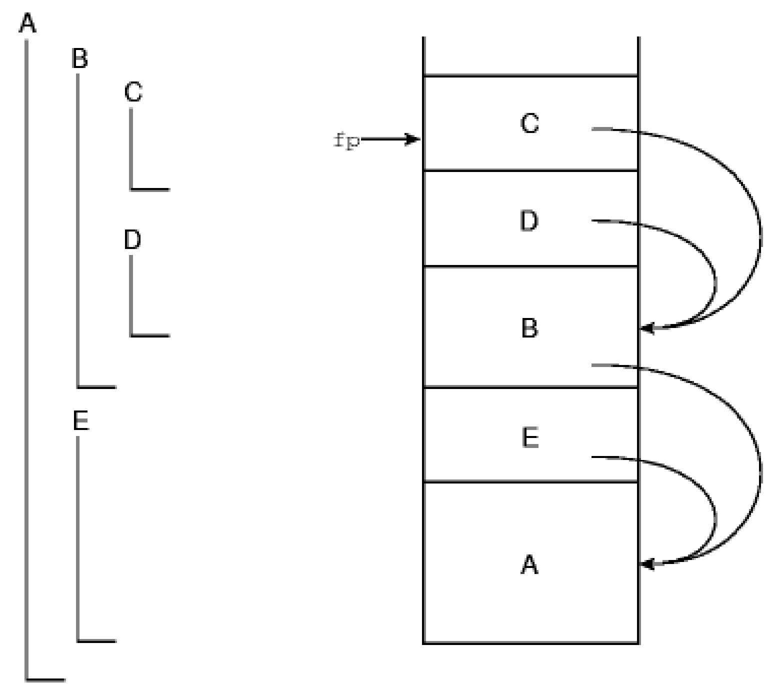
– Can F1 call P2?	yes
– Can P4 call F1?	yes
– Can P2 call F1?	no
– Can P3 use A1?	yes
– Can P3 use X?	yes
– Can P3 use A2?	yes
– If P4 uses X, what type is X?	real
– If F1 uses X, what type is X?	integer

```
procedure P1 (A1 : T1);
var X : real;
    ...
    procedure P2 (A2 : T2);
        ...
        procedure P3 (A3 : T3);
        ...
        begin
            ...        (* body of P3 *)
        end;
        ...
    begin
        ...            (* body of P2 *)
    end;
    ...
    procedure P4 (A4 : T4);
        ...
        function F1 (A5 : T5) : T6;
        var X : integer;
        ...
        begin
            ...        (* body of F1 *)
        end;
        ...
    begin
        ...            (* body of P4 *)
    end;
    ...
begin
    ...                (* body of P1 *)
end
```

Static Scope

- Objects defined in the current scope can be found directly in the current (topmost) frame on the stack
- What about objects defined in outer scopes?

- Static chains:
  - Each frame contains a pointer (static link) to the frame of the subroutine inside which it was declared
  - Example: C is nested 2 levels deep inside A. From C, to find an object defined in A, one need to follow 2 links.



Static Scope

- In C nested functions are not allowed
- However, there can still be nested scopes. How?
  - a new scope is defined any time { } are used
  - variables declared inside { } are local to that scope

```
{
   int x;
   {
                        float x, y;
                        ...
   }
   ...
}
```

Static Scope

- Another example of static scope rules is the import/export strategy used in modules
- A module is used for information hiding. It encapsulates a collection of objects (subroutines, variables, types, etc) so that:
  - objects inside are visible to each other
  - objects inside are not visible outside unless explicitly exported
  - objects outside are not visible inside unless explicitly imported (in general)

Static Scope

- Examples of languages with modules:
  - Clu (clusters)
  - Modula (modules)
  - Turing
  - Ada (packages)

- Closed scopes - scopes into which names must be explicitly imported (in Modula, Euclid)
- Open scopes - scopes where imports are automatic (in Ada)

- Subroutine scopes can also be open (usually) or closed (in Euclid)

Static Scope

- A module (manager for stacks) in Modula-2:

```
CONST stack_size = ...
TYPE element = ...
...
MODULE stack_manager;
IMPORT element, stack_size;
EXPORT stack, init_stack, push, pop;
TYPE
    stack_index = [1..stack_size];
    STACK = RECORD
        s : ARRAY stack_index OF element;
        top : stack_index;          (* first unused slot *)
    END;

PROCEDURE init_stack (VAR stk : stack);
BEGIN
    stk.top := 1;
END init_stack;

PROCEDURE push (VAR stk : stack; elem : element);
BEGIN
    IF stk.top = stack_size THEN
        error;
    ELSE
        stk.s[stk.top] := elem;
        stk.top := stk.top + 1;
    END;
END push;

PROCEDURE pop (VAR stk : stack) : element;
BEGIN
    IF stk.top = 1 THEN
        error;
    ELSE
        stk.top := stk.top - 1;
        return stk.s[stk.top];
    END;
END pop;

END stack;
```

```
var A, B : stack;
var x, y : element;
...
init_stack (A);
init_stack (B);
...
push (A, x);
...
y := pop (B);
```

Dynamic Scope

- Recall that the key idea in static scope rules is that bindings are defined by the lexical structure of the program

- Dynamic scope
  - Bindings depend on the current state of program execution
  - To resolve a reference, choose the <u>most recent active binding</u> for that name encountered during execution
  - Typically used in interpreted languages

- Examples: APL, Snobol, early Lisp

Dynamic Scope

- Example - static vs. dynamic scope rules

```
a : integer
procedure first
    a := 1
procedure second
    a : integer
    first()
// main program
a := 2
second()
write(a)
```

- What is written if the scoping rules are:
  - static? 1
  - dynamic? 2
- If static scoping - a in procedure first refers to the global variable a (as there is no local declaration of a in first). Therefore, the global a is changed to 1
- If dynamic scoping - a in procedure first refers to the local variable a declared in procedure second (this is the last binding for a encountered at run time, as first is called from second). Therefore, the local a is changed to 1, and then destroyed when returning from second

Binding Rules

- Recall that a referencing environment represents the set of active bindings at a given moment at run time
  - Corresponds to a collection of scopes that are examined (in order) to find a binding
  - Scope rules determine that collection and its order
- Additional issue when a subroutine is passed as a parameter, returned from another subroutine, stored into a variable:
  - When the function is called, what referencing environment will it use?
- Binding rules:
  - Shallow binding - use the environment from the moment of function call
  - Deep binding - use the environment from the moment when function was passed/returned/stored

Binding Rules

- Shallow binding
  - When the function is called, the current referencing environment (at call time) is used
  - Advantage: ease of implementation
  - Disadvantage: hard to understand, may alter programmer's intention
  - Typically encountered in languages with dynamic scoping
  - Examples: early Lisp, Snobol

Binding Rules

- Deep binding
  - When the function is passed/returned/stored, the current referencing environment and the function itself are packed together and called a closure
  - When the function is called, the environment stored in the closure (corresponding to the moment when function was passed/returned/stored) is used
  - Advantage: more intuitive for programmer
  - Disadvantage: harder to implement - need to save the referencing environment
  - Examples: Scheme, Algol, Pascal

Binding Rules

- Shallow vs. deep binding

```
procedure C; begin end;

procedure A (P : procedure; i : integer);
   procedure B;
   begin B
      write(i);
   end B;
begin A
   if i = 1 then A(B,2)
   else P;
end A;

begin main
   A(C,1);
end main.
```

- What is written in the case of:
  - deep binding? 1
  - shallow binding? 2

Binding Rules

- The binding rules (deep or shallow binding) are irrelevant unless you pass procedures as parameters, return them from functions, or store them in variables
- The difference will be noticeable only for references that are neither local nor global
  - Consequently, binding rules aren't relevant in languages such as C which have no nested subroutines
- To the best of our knowledge, no language with static scope rules has shallow binding

Announcements

- Readings
  - Rest of Chapter 3

Symbol Tables

- Symbol table
  - Used to keep track of names (and what they refer to) in a statically scoped language
  - Built and used during compilation
- Basic idea
  - Implement as a dictionary - maps names to the information the compiler knows about them (type, etc)
  - Operations - insert and lookup
- How to handle nested scopes?
  - Problem: a local declaration can hide a global one with the same name
  - Cannot remove the global one – because it becomes visible again outside the local scope

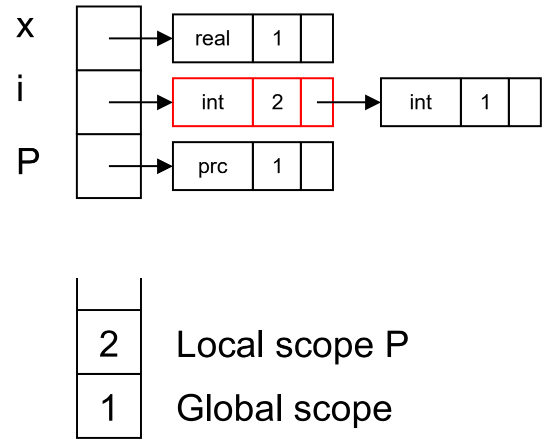
Symbol Tables

- Solution (LeBlanc-Cook symbol table)
  - Use scope labels and a separate stack of active scopes
  - When a new scope is encountered (at compilation)
    - assign a label to it
    - push an entry for that scope on the stack (enter_scope)
  - When a declaration is encountered
    - insert the name in the table together with the label of current scope
  - When a name is referenced
    - lookup for the name (in the table), that has the label of the current or outer scopes (as shown in the stack)
  - When a leaving a scope
    - pop the scope from the stack (leave_scope)
  - All names are kept in the table, nothing is ever deleted
  - Only entries on the stack are pushed and popped

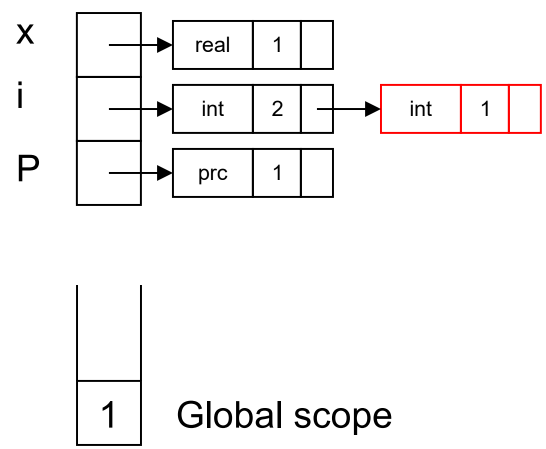
Symbol Tables

- Example - LeBlanc-Cook symbol table

x : real
i : integer
procedure P
    i : integer
    i := 4   ← compiler is here
// main program
i := 3



x : real
i : integer
procedure P
    i : integer
    i := 4
// main program
i := 3   ← compiler is here



Association Lists

- Two approaches for accessing names in a dynamically scoped language (at run time):
  - Association list
  - Central reference table
- Association list
  - a stack of pairs name / information about it
  - when a declaration is encountered (at execution), push it on top of stack
  - when a name is referenced, search in the stack from the top down, until found
  - dynamic scoping - first occurrence in stack corresponds to last declaration (execution time)
  - when a leaving a scope, pop all local bindings from the stack
  - problem: if a name has been declared long ago, it is buried deep in the stack

Central Reference Tables

- Central reference table
  - keep a central table (dictionary) with a slot for each name
  - at each slot keep an association list (stack) for that name
  - faster to lookup - search only in the stack corresponding to that name

Overloading and Related Concepts

- So far we have assumed that every name refers to one object in a given scope
- Not always the case - sometimes, a name may refer to more than one object in a scope
- Semantic rules need to infer which binding is intended

- Several variants:
  - overloading
  - coercion
  - polymorphism
  - generics

Overloading

- Overloading
  - implement several objects (typically functions) with the same name
  - the compiler infers the correct binding based on context
    - for functions, they must differ in the number or types of arguments
- Some overloading happens in almost all languages
  - + for integers vs + for floats
  - read and write in Pascal
- Example in C++:

```
struct complex {
    double real, imaginary;
};
enum base {dec, bin, oct, hex};

int i;
complex x;

void print_num (int n) ...
void print_num (int n, base b) ...
void print_num (complex c) ...

print_num (i);      // uses the first function above
print_num (i, hex); // uses the second function above
print_num (x);      // uses the third function above
```

Coercion

- Coercion
  - the process of automatically converting an object of one type into an object of another type, when the second type is expected
- Example in C:

```
void f (float x)
{ ... }

f(5);
```

- Pascal – limited number of coercions
- C++ – extremely rich set of coercions, allows programmer to define more
- Ada – no coercions

Polymorphism

- Polymorphism
  - used when passing parameters to functions
  - the types of the parameters must have some characteristics in common, and the function must use only those characteristics
  - there is only one function (unlike overloading)
  - nothing is converted (unlike coercion)

- Examples:
  - A function that computes absolute value (abs) can be written for any type that provides 2 operations: “comparison to zero” and “negation”
  - In Scheme – a function that computes the number of elements in a list. The elements in the list can have any type, as long as there is a "successor" operation and a “null?” test

Generics

- Generic subroutine/module
  - represents a template that can be used to create multiple concrete subroutines/modules, that differ in minor ways
  - the template definition is parameterized
  - when using the template, an actual value is specified for the parameter

- Example:
  - In C++ – define a template that implements a generic queue containing elements of type <T>
  - The template can then be used to declare queues of integers, floats, strings, various structures, etc.

Announcements

- Readings
  - Chapter 6