3 Names, Scopes, and Bindings

3.10 Solutions Manual

This manual contains suggested solutions to many of the PLP exercises. It is provided *only* to instructors who have adopted the text in their course.

3.1 Indicate the binding time (e.g., when the language is designed, when the program is linked, when the program begins execution, etc.) for each of the following decisions in your favorite programming language and implementation. Explain any answers you think are open to interpretation.

- The number of built-in functions (math, type queries, etc.)
- The variable declaration that corresponds to a particular variable reference (use)
- The maximum length allowed for a constant (literal) character string
- The referencing environment for a subroutine that is passed as a parameter
- The address of a particular library routine
- The total amount of space occupied by program code and data

Answer: Here are answers for C:

The number of built-in functions is originally bound at language design time, though it may be increased by certain implementations. C has just a few functions that are truly built-in, notably `sizeof`. A large number of additional functions are defined by the standard library. Several of these, including `printf`, `malloc`, `assert`, and the various `stdarg` routines, are often special-cased by the compiler in order to generate faster or safer code.

The variable declaration that corresponds to a particular variable reference (use) is bound at compile time: C uses static scope.

The maximum length of a character string (if there is a limit) is bound at language implementation time.

Because C does not have nested subroutines, the referencing environment for a subroutine that is passed as a parameter is always the same as the environment in effect when the subroutine was declared.

The address of a particular library function is bound by the linker in most systems, though it may not be known until load time or even run time in systems that perform dynamic linking (Section 14.7). Note that we're speaking here of virtual addresses; physical addresses are invisible to the running program, and are often changed by the operating system during execution).

The total amount of space occupied by program code and data is bound at run time: the amount of stack and heap space needed will often depend on the input.

3.2 In Fortran 77, local variables are typically allocated statically. In Algol and its descendants (e.g., Pascal and Ada), they are typically allocated in the stack. In Lisp they are typically allocated at least partially in the heap. What accounts for these differences? Give an example of a program in Pascal or Ada that would not work correctly if local variables were allocated statically. Give an example of a program in Scheme or Common Lisp that would not work correctly if local variables were allocated on the stack.

Answer: Fortran 77 lacks recursion, so there can never be more than one live instance of the local variables of a given subroutine. Algol and its descendants require a stack to accommodate multiple copies. The following would not work correctly with statically allocated local variables:

```
function sum(f, low, high) if low = high return f(low) else return f(low) + sum(f, low + 1, high)
```

Function `sum` needs to remember the value of `low` during the recursive call. (As we shall see in Section 6.6.1, it is possible to write a *tail recursive* version of `sum` that does not need to remember anything during the recursive call, but only if we are willing to change the calling signature of the function or to exploit the associativity of addition.)

Algol and its descendants (for the most part) have limited extent for local variables, meaning that the values of those variables are lost when control leaves the scope in which they were declared. Lisp and its descendants require allocation in the heap to accommodate unlimited extent. The following would not work correctly with stack-allocated variables:

```
function add_n(n) return { function(k) return n + k }
```

The intent here is to return a function which, when called, will add to its argument `k` the value `n` originally passed to `add_n`. That value (`n`) must remain accessible as long as the function returned by `add_n` remains accessible.

3.3 Give two examples in which it might make sense to delay the binding of an implementation decision, even though sufficient information exists to bind it early.

Answer: There are many possible answers. Here are a few:

Just-in-time compilation allows a system to minimize code size, obtain the most recent implementations of standard abstractions, and avoid the overhead of compiling functions that aren't used. Dynamic linking (Section 14.7) has similar advantages over static linking. (Just-in-time compilation also allows us to ship machine-independent code around the Internet, but in that case we don't have enough information to bind early.)

Local variables in Fortran 77 may be allocated on the stack, rather than in static memory, in order to minimize memory footprint and to facilitate interoperability with standard tools (e.g., debuggers).

Various aspects of code generation may be delayed until link time in order to facilitate whole-program code improvement.

3.4 Give three concrete examples drawn from programming languages with which you are familiar in which a variable is live but not in scope.

Answer: Here are a few possibilities:

(a) In Ada, if procedure `P` declares a local variable named `x`, then a global variable also named `x` will be live but not in scope when executing `P`.

(b) In Modula-2, a global variable declared in a module is live but not in scope when execution is not inside the module.

(c) In C, a static variable declared inside a function is live but not in scope when execution is not inside the function.

(d) in C++, non-public fields of an object of class `C` are live but not in scope when execution is not inside a method of `C`.

3.5 Consider the following pseudocode, assuming nested subroutines and static scope:

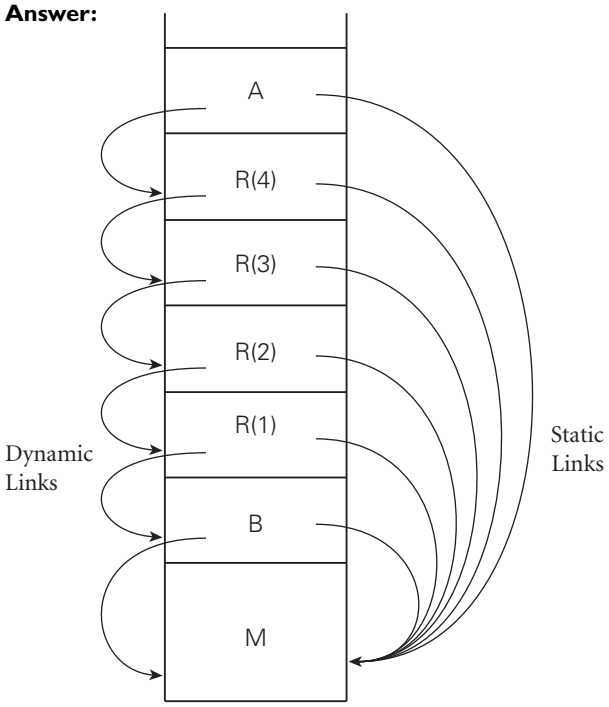
```
procedure main
    g : integer

    procedure B(a : integer)
        x : integer

        procedure A(n : integer)
            g := n

        procedure R(m : integer)
            write_integer(x)
            x /:= 2 -- integer division
            if x > 1
                R(m + 1)
            else
                A(m)

        -- body of B
        x := a × a
        R(1)

    -- body of main
    B(3)
    write_integer(g)
```

(a) What does this program print?

Answer: 9 4 2 1 4.

(b) Show the frames on the stack when A has just been called. For each frame, show the static and dynamic links.

Answer:



(c) Explain how A finds g.

Answer: It dereferences its static link to find the stack frame of B. Within this frame it finds B's static link at a statically known offset. It dereferences that to find the stack frame of main, within which it finds g, again at a statically known offset. (This assumes that g is a local variable of main, *not* a global variable.)

3.6 As part of the development team at MumbleTech.com, Janet has written a list manipulation library for C that contains, among other things, the code in Figure 3.18.

(a) Accustomed to Java, new team member Brad includes the following code in the main loop of his program:

```
list_node *L = 0;
while (more_widgets()) {
    insert(next_widget(), L);
}
L = reverse(L);
```

```
typedef struct list_node {
    void *data;
    struct list_node *next;
} list_node;

list_node *insert(void *d, list_node *L) {
    list_node *t = (list_node *) malloc(sizeof(list_node));
    t->data = d;
    t->next = L;
    return t;
}

list_node *reverse(list_node *L) {
    list_node *rtn = 0;
    while (L) {
        rtn = insert(L->data, rtn);
        L = L->next;
    }
    return rtn;
}

void delete_list(list_node *L) {
    while (L) {
        list_node *t = L;
        L = L->next;
        free(t->data);
        free(t);
    }
}
```

Figure 3.18 List management routines for Exercise 3.6.

Sadly, after running for a while, Brad's program always runs out of memory and crashes. Explain what's going wrong.

Answer: The `reverse_list` routine produces a new list, composed of new list nodes. When Brad assigns the return value back into `L` he loses track of the old list nodes, and never reclaims them. In other words, his program has a memory leak. After a number of iterations of his main loop, Brad has exhausted the heap and his program can't continue.

(b) After Janet patiently explains the problem to him, Brad gives it another try:

```
list_node *L = 0;
while (more_widgets()) {
    insert(next_widget(), L);
}
list_node *T = reverse(L);
delete_list(L);
L = T;
```

This seems to solve the insufficient memory problem, but where the program used to produce correct results (before running out of memory), now its output is strangely corrupted, and Brad goes back to Janet for advice. What will she tell him this time?

Answer: While the call to `delete_list` successfully reclaims the old list nodes, it also reclaims the widgets. The new, reversed list thus contains dangling references. These refer to locations in the heap that may be used for newly allocated data, which may be corrupted by uses of the elements in the reversed list. Brad seems to have been lucky in that he isn't corrupting the heap itself (maybe his widgets are the same size as list nodes), but without Janet's help he may have a lot of trouble figuring out why widgets are changing value "spontaneously."

3.7 Rewrite Figures 3.7 and 3.8 in C.

Answer: Since encapsulation in C depends on separate compilation, we give multi-file solutions. First, the module-as-abstraction equivalent of Figure 3.7. Header file `stack.h`:

```
#define STACK_SIZE  100         /* your size here */
typedef int element;            /* your type here */

extern void push(element);
extern element pop();
```

Implementation file `stack.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

typedef short stack_index;

element s[STACK_SIZE];
stack_index top = 0;        /* first unused slot */

static void error(const char* err) {
    fprintf(stderr, "stack error: %s\n", err);
    exit(1);
}

void push(element elem) {
    if (top == STACK_SIZE) error("overflow");
    else s[top++] = elem;
}

element pop() {
    if (top == 0) error("underflow");
    else return s[--top];
}
```

Client file:

```
#include <stdio.h>
#include "stack.h"

int main() {
    element x, y;

    x = 3;
    push(x);

    y = pop();
    printf("y == %d\n", y);
}
```

Now the module-as-manager equivalent of Figure 3.8. Header file `stack_manager.h`:

```
#define STACK_SIZE  100         /* your size here */
typedef int element;            /* your type here */

typedef short stack_index;      /* should not be used by clients */

typedef struct {
    element s[STACK_SIZE];
    stack_index top;
} stack;

extern void init_stack(stack*);
extern void push(stack*, element);
extern element pop(stack*);
```

Implementation file `stack_manager.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include "stack_manager.h"

static void error(const char* err) {
    fprintf(stderr, "stack error: %s\n", err);
    exit(1);
}

extern void init_stack(stack* stk) {
    stk->top = 0;
}

extern void push(stack* stk, element elem) {
    if (stk->top == STACK_SIZE) error("overflow");
    else stk->s[stk->top++] = elem;
}
```

```
extern element pop(stack* stk) {
    if (stk->top == 0) error("underflow");
    else return stk->s[--stk->top];
}
```

Client file:

```
#include <stdio.h>
#include "stack_manager.h"

int main() {
    element x, y;

    stack A, B;

    init_stack(&A);
    init_stack(&B);

    x = 3;
    push(&A, x);
    push(&B, pop(&A));

    y = pop(&B);
    printf("y == %d\n", y);
}
```

3.8 Modula-2 provides no way to divide the header of a module into a public part and a private part: everything in the header is visible to the users of the module. Is this a major shortcoming? Are there disadvantages to the public/private division (e.g., as in Ada)? (For hints, see Section 9.2.)

Answer: As discussed in the sidebar in Section 9.1 ("What goes in a class declaration?"), the "private" part of a module header or class declaration provides information that, while not semantically part of the interface of the abstraction, is nonetheless required by the compiler in order to generate code for clients of the abstraction. There are two problems this notion. First, it violates the spirit of information hiding, because it allows the writers of client code to see inside the abstraction, even if they can't *use* what they see. Second, it forces the creator of the abstraction to think about, and explicitly identify, the extra information needed by the compiler.

As described in Section 9.2.1 ("Making Do Without Module Headers"), Java and C# take an alternative approach, dispensing with headers altogether. The creator of an abstraction identifies the semantically public aspects of the interface. The compiler examines the full implementation to find whatever else it needs.

3.9 Consider the following fragment of code in C:

```
{   int a, b, c;
    ...
    {   int d, e;
        ...
        {   int f;
            ...
        }
        ...
    }
    ...
    {   int g, h, i;
        ...
    }
    ...
}
```

Assume that each integer variable occupies four bytes. How much total space is required for the variables in this code? Describe an algorithm that a compiler could use to assign stack frame offsets to the variables of arbitrary nested blocks, in a way that minimizes the total space required.

Answer: In the code above, variables `a`, `b`, and `c` are live throughout the execution of the outer block. Variables `d`, `e`, and `f` are needed only in the first nested block, and can overlap the space devoted to `g`, `h`, and `i`. A total of $4 \times 6 = 24$ bytes is required.

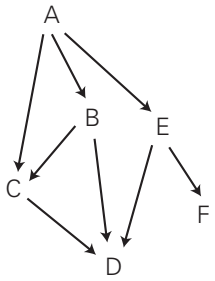
When compiling a subroutine, the compiler can construct a tree in which each node represents a block, and is a child of the node that represents the surrounding block. Variables declared in the outermost block are assigned locations at the beginning of the subroutine's space. Variables in a nested block are assigned locations immediately following the variables of the surrounding (parent) block. Variables of siblings overlap.

3.10 Consider the design of a Fortran 77 compiler that uses static allocation for the local variables of subroutines. Expanding on the solution to the previous question, describe an algorithm to minimize the total space required for these variables. You may find it helpful to construct a *call graph* data structure in which each node represents a subroutine and each directed arc indicates that the subroutine at the tail may sometimes call the subroutine at the head.

Answer: Since Fortran 77 lacks recursion, the call graph for a Fortran 77 program is guaranteed to be acyclic. If two subroutines never appear on the same path from `main` in this graph, then their local variables will never be needed at the same time, and may in fact share space.

If our compiler allocates local variables statically, we can choose a minimum-size allocation by performing a topological sort of the call graph. The "frame" of any subroutine that can be called *only* from the main program is placed at address 0. After this, the frame of any subroutine that can be called only by subroutines whose frames have already been assigned locations is placed at the first address that is beyond the frames of all of its potential callers.

As an example, consider the acyclic call graph below. It shows that subroutine `E` is never active at the same time as `B` or `C`. When allocating space we can therefore overlap `E`'s storage with that of `B` and `C`. Similarly, the space for `D` and `F` can overlap, though both must be disjoint from `A` and `E`. (The vertical offsetting of boxes is for clarity of presentation only.)



3.11 Consider the following pseudocode:

```
procedure P(A, B : real)
    X : real

    procedure Q(B, C : real)
        Y : real
        . . .

    procedure R(A, C : real)
        Z : real
        . . .                          -- (*)
    . . .
```

Assuming static scope, what is the referencing environment at the location marked by (*)?

Answer: P, B, X, Q, R, A (parameter to R), C (parameter to R), and Z. Note that the parameters and local variable of Q are not in scope, nor is P's parameter A.

3.12 Write a simple program in Scheme that displays three different behaviors, depending on whether we use `let`, `let*`, or `letrec` to declare a given set of names. (Hint: to make good use of `letrec`, you will probably want your names to be functions [`lambda` expressions].)

Answer:

```
(let ((a (lambda (n) 1))
      (b (lambda (n) 2)))
  (let ((a (lambda (n) (if (zero? n) 3 (b (- n 1)))))
        (b (lambda (n) (if (zero? n) 4 (a (- n 1))))))
    (list (a 5) (b 5))))
```

The standard `let` evaluates the second element of all tuples before creating any new names, so the inner `a` and `b` both call the outer, constant functions, and the code evaluates to to `(2 1)`. If we replace the inner `let` with `let*`, then the inner `a` refers to the outer `b`, but the inner `b` refers to the *inner* `a`, and the code evaluates to `(2 2)`. If we replace the inner `let` with `letrec`, then the inner `a` and `b` refer to each other, and the code evaluates to `(4 3)`.

3.13 Consider the following pseudocode:

```
x : integer       -- global

procedure set_x(n : integer)
    x := n

procedure print_x
    write_integer(x)

procedure first
    set_x(1)
    print_x

procedure second
    x : integer
    set_x(2)
    print_x

set_x(0)
first()
print_x
second()
print_x
```

What does this program print if the language uses static scoping? What does it print with dynamic scoping? Why?

Answer: With static scoping it prints `1 1 2 2`. With dynamic scoping it prints `1 1 2 1`. The difference lies in whether set_x sees the global x or the x declared in second when it is called from second.

3.14 Consider the programming idiom illustrated in Example 3.20. One of the reviewers for this book suggests that we think of this idiom as a way to implement a central reference table for dynamic scope. Explain what is meant by this suggestion.

Answer: The standard way of describing dynamic scope mirrors its implementation with an association list. The reviewer's suggestion mirrors implemtation with central reference table, as described in the sidebar ("Thinking about dynamic scope") in Section 13.4.1. Semantically the two descriptions are equivalent: we can say that the elaboration of a declaration creates a new dynamic variable that hides the old, or we can say that it saves the old value and introduces a new one (possibly undefined, depending on whether the declaration specifies a value). At end of scope either the old variable becomes visible again or the old value is restored; the effect is the same either way.

3.15 If you are familiar with structured exception-handling, as provided in Ada, Modula-3, C++, Java, C#, ML, Python, or Ruby, consider how this mechanism relates to the issue of scoping. Conventionally, a `raise` or `throw` statement is thought of as referring to an exception, which it passes as a parameter to a

handler-finding library routine. In each of the languages mentioned, the exception itself must be declared in some surrounding scope, and is subject to the usual static scope rules. Describe an alternative point of view, in which the `raise` or `throw` is actually a reference to a *handler*, to which it transfers control directly. Assuming this point of view, what are the scope rules for handlers? Are these rules consistent with the rest of the language? Explain. (For further information on exceptions, see Section 8.5.)

Answer: If we think of a `raise`/`throw` statement as containing a reference to a handler, then entry into a code block with a handler for exception *E* is very much like the elaboration of a dynamically scoped declaration. The new handler hides any previous handler for the same exception. The old handler "declaration" becomes visible again when execution leaves the nested scope.

Given that all the languages listed in the question use static scoping for all other names, the dynamic scoping explanation of handlers is not particularly appealing—hence the usual description in terms of static scoping within subroutines and "an exceptional return" across subroutines. Like the two descriptions of dynamic scope (Exercise 3.14), however, the two ways of modeling exceptions are equivalent.

3.16 Consider the following pseudocode:

```
x : integer       -- global

procedure set_x(n : integer)
    x := n

procedure print_x
    write_integer(x)

procedure foo(S, P : function; n : integer)
    x : integer := 5
    if n in {1, 3}
        set_x(n)
    else
        S(n)
    if n in {1, 2}
        print_x
    else
        P

set_x(0); foo(set_x, print_x, 1); print_x
set_x(0); foo(set_x, print_x, 2); print_x
set_x(0); foo(set_x, print_x, 3); print_x
set_x(0); foo(set_x, print_x, 4); print_x
```

Assume that the language uses dynamic scoping. What does the program print if the language uses shallow binding? What does it print with deep binding? Why?

Answer: With shallow binding, `set_x` and `print_x` always access `foo`'s local `x`. The program prints

```
1 0  2 0  3 0  4 0
```

With deep binding, `set_x` accesses the global `x` when `n` is even and `foo`'s local `x` when `n` is odd. Similarly, `print_x` accesses the global `x` when `n` is 3 or 4 and `foo`'s local `x` when `n` is 1 or 2. The program prints

```
1 0  5 2  0 0  4 4
```

3.17 Consider the following pseudocode:

```
x : integer := 1
y : integer := 2

procedure add
    x := x + y

procedure second(P : procedure)
    x : integer := 2
    P()

procedure first
    y : integer := 3
    second(add)

first()
write_integer(x)
```

(a) What does this program print if the language uses static scoping?

(b) What does it print if the language uses dynamic scoping with deep binding?

(c) What does it print if the language uses dynamic scoping with shallow binding?

Answer: (a) 3; (b) 4; (c) 1.

3.18 In Section 3.6.3 we noted that while a single `min` function in Fortran would work for both integer and floating-point numbers, overloading would be more efficient, because it would avoid the cost of type conversions. Give an example in which overloading does not seem advantageous—one in which it makes more sense to have a single function with floating-point parameters, and perform coercion when integers are supplied.

Answer: The following function in C uses Heron's formula to compute the area of a triangle with sides of length `a`, `b`, and `c`:

```
double area(double a, double b, double c) {
    double s = (a + b + c) / 2;
    return sqrt(s * (s - a) * (s - b) * (s - c));
}
```

Here `s` has a 50/50 chance of being nonintegral even if `a`, `b`, and `c` are all integers, and the result is almost certainly nonintegral, so integer-to-floating-point conversions are almost certainly going to be needed in any case. Moreover the possibility that any subset of `{a, b, c}` might be integral means that eight different functions would be needed to cover all combinations, which seems excessive.

3.19 (a) Write a polymorphic sorting routine in Scheme.

Answer:

```
;; quicksort in Scheme
;; will work for any type for which '<' is defined; i.e., any numeric type
(define sort
  (lambda (L)
    (letrec ((partition
              (lambda (e L A B)
                (if (null? L) (cons A B)
                    (let ((c (car L)))
                      (if (< c e)
                          (partition e (cdr L) (cons c A) B)
                          (partition e (cdr L) A (cons c B))))))))
      (cond
        ((null? L) L)
        ((null? (cdr L)) L)
        (else (let* ((AB (partition (car L) (cdr L) '() '()))
                     (A (car AB))
                     (B (cdr AB)))
                (append (sort A)
                        (list (car L))
                        (sort B))))))))
```

(b) Write a generic sorting routine in C++, Java, or C#. (For hints, see Section 8.4.)

Answer:

```
// insertion sort in Java
public static <T extends Comparable> void sort(T A[]) {
    // interface Comparable is defined in the Java standard library;
    // it supports the compareTo method
    for (int i = 1; i < A.length; i++) {
        // A[0..i-1] is sorted
        int j = i;
        T t = A[i];
        for (; j > 0; j--) {
            if (t.compareTo(A[j-1]) >= 0) break;
            A[j] = A[j-1];
        }
```

```
        A[j] = t;
        // A[0..i] is sorted
    }
}
```

(c) Write a nongeneric sorting routine using subtype polymorphism in your favorite object-oriented language. Assume that the elements to be sorted are members of some class derived from class `ordered`, which has a method `precedes` such that `a.precedes(b)` is true if and only if `a` comes before `b` in some canonical total order. (For hints, see Section 9.4.)

Answer:

```
// insertion sort in C++

class ordered {
public:
    virtual bool precedes(ordered&) = 0;
};

// example concrete class derived from ordered:
class Integer : public ordered {
public:
    int val;
    Integer(int i) : val(i) { };
    virtual bool precedes(ordered& other) {
        Integer o = dynamic_cast<Integer&>(other);
            // in the absence of generics, this cast is inevitable
        return val < o.val;
    }
};

void sort (ordered* A[], int A_size) {
    for (int i = 1; i < A_size; i++) {
        int j = i;  ordered* t = A[i];
        for (; j > 0; j--) {
            if (A[j-1]->precedes(*t)) break;
            A[j] = A[j-1];
        }
        A[j] = t;
    }
}

ordered* Integers[20];
...
sort (Integers, 20);
}
```