Chapter 3:: Names, Scopes, and Bindings

1

Programming Language Pragmatics

Michael L. Scott

1

Introduction

2

- high-level languages are abstracted away from the underlying architecture
  - machine independence
  - ease of programming/understanding
- naming, scope, and binding are important concepts in high-level languages

2

Name, Scope, and Binding

3

- a name is exactly what you think it is
  - most names are identifiers
  - symbols (like '+') can also be names
  - can refer to variables, constants, operations, types, etc.
  - can also aid in abstraction by modularizing a complicated sequence of code by a simple identifier (e.g., function name)

3

Name, Scope, and Binding

4

- 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
- a complete set of bindings at a given point in time in a program is termed the referencing environment
- binding time is the point at which a binding is created or, more generally, the point at which any implementation decision is made

4

Binding

5

- implementation decision times
  - language design time
    - program structure, possible types, constructors
  - language implementation time
    - I/O, arithmetic overflow, type equality, precision
  - program writing time
    - algorithms, names
  - compile time
    - plan for data layout
  - link time
    - layout of program in memory, inclusion of external modules
  - load time
    - choice of physical addresses

5

Binding

6

- implementation decision times (continued):
  - run time
    - value/variable bindings, sizes of strings
    - includes
      - 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

6

Binding

7

- binding times very important in programming languages
  - *static* binding occurs before run time
  - *dynamic* binding occurs at run time
- compiled languages tend to have early binding times
  - associated with greater efficiency
- interpreted languages tend to have later binding times
  - associated with greater flexibility
  - some languages difficult to compile because their definitions require late binding decisions

7

Binding

8

- scope rules control bindings
  - fundamental to all programming languages is the ability to name data
    - 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

8

Lifetime and Storage Management

9

- lifetime: period of time from creation to destruction of a binding
- lifetime 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

9

Lifetime and Storage Management

10

- garbage: object outlives binding
- dangling reference: binding outlives object
- scope: region of the program in which the binding is *active*

10

Lifetime and Storage Management

11

- storage allocation mechanisms
  - static
  - stack
  - heap
- static allocation for
  - code
  - global variables
  - static variables
    - present for entire execution of program
  - explicit constants (including strings, sets, etc.)
    - e.g., "`hello`"
  - some values stored in instructions
    - e.g., `add $3,%eax`

11

Lifetime and Storage Management

12

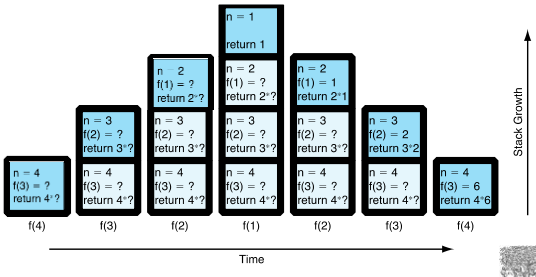
- runtime stack for
  - parameters
  - local variables
  - temporaries
- why a stack?
  - allocate space for recursive routines
  - reuse space
  - maintain ordering of functions
    - currently running function at top of stack
    - each time a function is called, a stack frame for the function is pushed on the stack

12

Lifetime and Storage Management

13

- call stack for factorial of 4



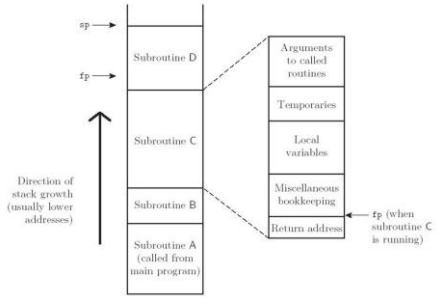
Lifetime and Storage Management

14

- contents of a stack frame
  - arguments and returns
  - local variables
  - temporaries
  - bookkeeping (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

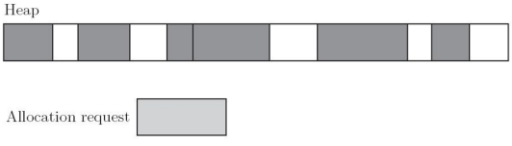
15



Lifetime and Storage Management

16

- the heap is used for dynamic allocation
  - memory requested during runtime



- memory request
  - shaded blocks are in use
  - clear blocks are free
  - enough space overall, but not contiguous

Scope Rules

17

- scope: program section of maximal size in which no bindings change, or at least in which no re-declarations are permitted
- in most languages with subroutines (functions), 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
- on subroutine exit
  - destroy bindings for local variables
  - reactivate bindings for global variables that were deactivated
  - exception: static variables in C

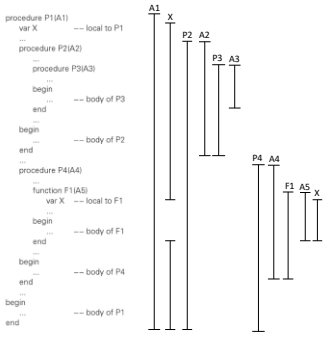
Scope Rules

18

- with static 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 (e.g., C and C++) employ static scope rules
- C/C++ functions and variables are not defined until seen by the compiler
  - scope begins at declaration and continues to end of current scope

Scope Rules

19



19

Scope Rules

20

- classical example of static scope rules is the most closely nested rule
  - 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
- classes in object-oriented languages have even more sophisticated (static) scope rules

20

Scope Rules

21

```
var a : integer;   { global variable }

procedure first();
begin
   a := 1;
end;

procedure second();
var a : integer;
begin
   first();
end;

begin
   a := 2;
   second();
   write(a);
end.
```


21

Scope Rules

22

- 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
    - correct means closest to the top of the stack
  - 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

22

Scope Rules

23

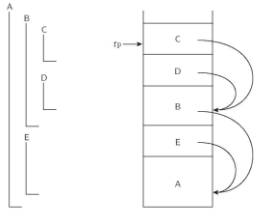


Figure 3.5: **Static chains.** Subroutines A, B, C, D, and E are nested as shown on the left. If the sequence of nested calls at run time is A, E, B, D, and C, then the static links in the stack will look as shown on the right. The code for subroutine C can find local objects at known offsets from the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

23

Scope Rules

24

- static scope rules
  - bindings are defined by the physical (lexical) structure of the program
- 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

24

Scope Rules

25

- 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

25

Scope Rules

26

```
var a : integer;   { global variable }

procedure first();
begin
   a := 1;
end;

procedure second();
var a : integer;
begin
   first();
end;

begin
   a := 2;
   second();
   write(a);
end.
```


26

Scope Rules
Example: Static vs. Dynamic

27

- if static scope rules are in effect (as would be the case in Pascal), the program prints **1**
- if dynamic scope rules are in effect, the program prints **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**
  - static scope rules require that the reference resolve to the most recent, compile-time binding, namely the global variable **a**

27

Scope Rules
Example: Static vs. Dynamic

28

- 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

28

Scope Rules
Example: Static vs. Dynamic

29

- 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
  - using dynamic scope, we modify the variable local to procedure **second**, not the global variable
  - in main, we write the global variable because the variable **a** local to procedure **second** is no longer active

29

Binding of Referencing Environments

30

- two methods for 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

30

Binding of Referencing Environments

- two methods for 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

- (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

- the 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
- 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

Symbol Table

- a *symbol table* is a data structure kept by a translator that allows it to keep track of each declared name and its binding
  - assume for now that each name is unique within its local scope
  - the data structure can be any implementation of a dictionary, where the name is the key

Source: Tucker & Noonan (2007)

Symbol Table

1. each time a scope is entered, push a new dictionary onto the stack
2. each time a scope is exited, pop a dictionary off the top of the stack
3. for each name declared, generate an appropriate binding and enter the name-binding pair into the dictionary on the top of the stack
4. given a name reference, search the dictionary on top of the stack
   a) if found, return the binding
   b) otherwise, repeat the process on the next dictionary down in the stack
   c) if name not found in any dictionary, report an error

Source: Tucker & Noonan (2007)

Symbol Table: Static Scoping


```
1 void sort (float a[ ], int size) {
2  int i, j;
3   for (i = 0; i < size; i++)  // i, size local
4     for (j = i + 1; j < size; j++)
5        if (a[j] < a[i]) { // a, i, j local
6            float t;
7            t = a[i];      // t local; a, i nonlocal
8            a[i] = a[j];
9            a[j] = t;
10      }
11 }
```

Source: Tucker & Noonan (2007)

Symbol Table: Static Scoping

37

- C program, stack of dictionaries at line 7:
  - `<t, 6>`
  - `<j, 2> <i, 2> <size,1> <a, 1>`
  - `<sort, 1>`
- at lines 4 and 11:
  - `<j, 2> <i, 2> <size,1> <a, 1>`
  - `<sort, 1>`

Source: Tucker & Noonan (2007)

37

Symbol Table: Static Scoping

38

```
1 int h, i;
2 void B(int w) {
3   int j, k;
4   i = 2*w;
5   w = w+1;
6   ...
7 }
8 void A (int x, int y) {
9   float i, j;
10  B(h);
11  i = 3;
12  ...
13 }
14 void main() {
15    int a, b;
16    h = 5; a = 3; b = 2;
17    A(a, b);
18    B(h);
19    ...
20 }
```

Source: Tucker & Noonan (2007)

38

Symbol Table: Static Scoping

39

1. outer scope:
   `<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>`
2. function B:
   `<w, 2> <j, 3> <k, 3>`
3. function A:
   `<x, 8> <y, 8> <i, 9> <j, 9>`
4. function main:
   `<a, 15> <b, 15>`

Source: Tucker & Noonan (2007)

39

Symbol Table: Static Scoping

40

- symbol table stack for function B:
  - `<w, 2> <j, 3> <k, 3>`
  - `<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>`
- symbol table stack for function A:
  - `<x, 8> <y, 8> <i, 9> <j, 9>`
  - `<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>`
- symbol table stack for function main:
  - `<a, 15> <b, 15>`
  - `<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>`

Source: Tucker & Noonan (2007)

40

Symbol Table: Dynamic Scoping

41

```
1 int h, i;
2 void B(int w) {
3   int j, k;
4   i = 2*w;
5   w = w+1;
6   ...
7 }
8 void A (int x, int y) {
9   float i, j;
10  B(h);
11  i = 3;
12  ...
13 }
14 void main() {
15    int a, b;
16    h = 5; a = 3; b = 2;
17    A(a, b);
18    B(h);
19    ...
20 }
```

Source: Tucker & Noonan (2007)

41

Symbol Table: Dynamic Scoping

42

- call history: `main(17)` → `A(10)` → `B`
- function — dictionary

`B`	`<w, 2> <j, 3> <k, 3>`
`A`	`<x, 8> <y, 8> <i, 9> <j, 9>`
`main`	`<a, 15> <b, 15>`
	`<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>`

- reference to `i(4)` resolves to `<i,9>` in `A`

Source: Tucker & Noonan (2007)

42

Symbol Table: Dynamic Scoping

43

- call history: `main(18) → B`
- function dictionary

```
B       <w, 2> <j, 3> <k, 3>
main    <a, 15> <b, 15>
        <h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
```

- reference to `i(4)` resolves to `<i,1>` in `A`

Source: Tucker & Noonan (2007)

43

The Meaning of Names within a Scope

44

- aliasing
  - aliases: two or more names that refer to the same object at the same point in the program
  - what are aliases good for?
    - space saving - modern data allocation methods are better
    - multiple representations - unions are better
    - linked data structures - legit
  - aliases arise in parameter passing as an unfortunate side effect
    - some scope rules are designed to prevent this

44

Aliasing

45

- consider the following example

```
double sum, sum_of_squares;
...
void accumulate(double& x) {
  sum += x;
  sum_of_squares += x * x;
}
...
accumulate(sum);
```

- easy to make a mistake by passing something accidentally
  - some languages make subroutines closed scopes
  - explicit import lists allow the compiler to detect when an alias is being created

45

Aliasing

46

- can make compiling more difficult to optimize

```
int a, b, *p, *q;
...
a = *p;
*q = 3;
b = *p;
```

- compiler wants to place `*p` in a register for possible future use
- can't use for `b` since `q` may be an alias

46

The Meaning of Names within a Scope

47

- overloading
  - some overloading happens in almost all languages
    - integer + vs. real +
    - read and write in Pascal
    - print in Python
  - some languages get into overloading in a big way
    - C++ (and hence Java and C#)

47

The Meaning of Names within a Scope

48

- overloading of enumeration constants in Ada

```
declare
   type month is (jan, feb, mar, apr, may, jun,
                  jul, aug, sep, oct, nov, dec);
   type print_base is (dec, bin, oct, hex);
   mo : month;
   pb : print_base;
begin
   mo := dec;      -- the month dec
   pb := oct;      -- the print_base oct
   print(oct);     -- error!  insufficient context to decide
```

  - must be able to provide context explicitly

```
print (month'(oct));
```

  - Modula-3 and C# require enumeration prefixes

```
mo := month.dec;
pb := print_base.oct;
```


48

Separate Compilation

49

- separately-compiled files in C provide a sort of *primitive modules environment*
  - rules for how variables work with separate compilation are messy
  - **`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
  - **`extern`** on a variable or function means that it is declared in another source file

49

Stack Diagram for Static Scoping

50

```
int h = 0, i = 1;

void B (int w) {
   int j = 2;
   static int k = 3;
   i = 2 * w;
   w = w + 1;
   k++;
   --j;
}

void A (int *x, int y) {
  int i = 1, j = 2;
  B(h);
  i = 3;
  *x = j + *x;
}
```

```
void main() {
     int a, b;
     h = 5; a = 3; b = 2;
     A(&a, b);
     B(b);
}
```

Source: Tucker & Noonan (2007)

50

Conclusions

51

- 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
- 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

51

C++ Addendum

52

- the following slides relate to object-oriented and functional languages

52

The Meaning of Names within a Scope

53

- 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
    - code takes a list of types, where the types have some commonality that will be exploited.
    - used in Ada and Smalltalk, primarily

53

The Meaning of Names within a Scope

54

- operator overloading in C++

```
class complex {
  double real, imaginary;
  …
  public:
     complex operator+ (complex other) {
        return complex (real + other.real,
        imaginary + other.imaginary);
     }
};
…
complex A, B, C;
…
C = A + B;
```


54

The Meaning of Names within a Scope

55

- it's worth distinguishing between some closely related concepts (part 2)
  - overloading
  - coercion
  - polymorphism

55

The Meaning of Names within a Scope

56

- overloading in Ada

```
function min(a, b : integer) return
  integer is ...
function min(x, y : real) return real
  is ...
```

- coercion in Fortran

```
real function min(x, y)
real x, y
...
```

  - types are automatically converted

56

The Meaning of Names within a Scope

57

- polymorphism
  - multiple forms
  - two types
    - parametric polymorphism
    - subtype polymorphism

57

The Meaning of Names within a Scope

58

- parametric polymorphism
  - generic functions: a syntactic template that can be instantiated in more than one way *at compile time*
    - create copies of the code
    - via macro processors in C
    - built-in in C++ (templates)
    - Clu
    - Ada

58

The Meaning of Names within a Scope

59

- explicit parametric polymorphism in Ada generics

```
generic
    type T is private;
    with function "<"(x, y : T) return Boolean;
function min(x, y : T) return T;

function min(x, y : T) return T is
begin
    if x < y then return x;
    else return y;
    end if;
end min;

function string_min is new min(string, "<");
function date_min is new min(date, date_precedes);
```


59

The Meaning of Names within a Scope

60

- implicit parametric polymorphism in Scheme

```
define min (lambda (a b) (if (< a b) a b)))
```

- implicit parametric polymorphism in Haskell

```
min a b = if a < b then a else b
```


60