Chapter 3:: Names, Scopes, and Bindings

Introduction

• “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

Name, Scope, and Binding

• 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., subroutine)

Name, Scope, and Binding

• 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

• binding time is the point at which a binding is created or, more generally, the point at which any implementation decision is made
• implementation decisions
  – language design time
    • program structure, possible types, constructors
  – language implementation time
    • I/O, arithmetic overflow, type equality (if unspecified in manual), precision

Binding

• implementation decisions (continued):
  – program writing time
    • algorithms, names
  – compile time
    • plan for data layout
  – link time
    • layout of whole program in memory, inclusion of external modules
  – load time
    • choice of physical addresses
• implementation decisions (continued):
  – run time
    • value/variable bindings, sizes of strings
    • subsuses
      – program start-up time
      – module entry time
      – elaboration time (point at which a declaration is first "seen")
      – procedure entry time
      – block entry time
      – statement execution time

• the terms static and dynamic are generally used to refer to things bound before run time and at run time, respectively
  – “static” is a coarse term; so is "dynamic"
  • IT IS DIFFICULT TO OVERSTATE THE IMPORTANCE OF BINDING TIMES IN PROGRAMMING LANGUAGES

• 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

• some languages difficult to compile because their definitions require late binding decisions
  – Smalltalk delays all type checking until run time
    • allows variable names to refer to objects of multiple types: polymorphism
    • allows very general code
• today we talk about the binding of identifiers to the variables they name

• 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

• 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

• 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

• 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

• 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

• some languages difficult to compile because their definitions require late binding decisions
  – Smalltalk delays all type checking until run time
    • allows variable names to refer to objects of multiple types: polymorphism
    • allows very general code
• today we talk about the binding of identifiers to the variables they name

• 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

• 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

• the period of time from creation to destruction is called the lifetime of a binding
  – if object outlives binding, it’s garbage
  – if binding outlives object, it’s 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

• storage allocation mechanisms
  – static
  – stack
  – heap
• static allocation for
  – code
  – globals
  – static or own variables
  – explicit constants (including strings, sets, etc.)
  – scalars may be stored in the instructions

Lifetime and Storage Management

• 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

Lifetime and Storage Management

• contents of a stack frame (Figure 3.2)
  – 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**

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

**Scope Rules**

- 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**

- 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

**Lifetime and Storage Management**

- the heap is used for dynamic allocation

**Scope Rules**

- 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**

- 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

• we will see classes - a relative of modules - later on, when discussing abstraction and object-oriented languages
  – 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

• 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

• 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
  – 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
• more details in Chapter 8

Scope Rules

• 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

Scope Rules

• 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

- 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

- 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

- 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

- static scope rules require that the reference resolve to the most recent, compile-time binding, namely the global variable a

Scope Rules
Example: Static vs. Dynamic

- 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
Binding of Referencing Environments

• 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

• 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

• (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

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

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

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

Symbol Table: Static Scoping

C program, stack of dictionaries at line 7:
- \(<t, 6>\)
- \(<j, 2> <i, 2> <size, 1> <a, 1>\)
- \(<\text{sort}, 1>\)

at lines 4 and 11:
- \(<j, 2> <i, 2> <size, 1> <a, 1>\)
- \(<\text{sort}, 1>\)

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 }

Symbol Table: Static Scoping

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 }

Symbol Table: Static Scoping

• symbol table stack for function B:
  - \(<w, 2> <j, 3> <k, 3>\)
  - \(<h, 1> <i, 1> <B, 2> <A, 8> <\text{main}, 14>\)
• symbol table stack for function A:
  - \(<x, 8> <y, 8> <i, 9> <j, 9>\)
  - \(<h, 1> <i, 1> <B, 2> <A, 8> <\text{main}, 14>\)
• symbol table stack for function main:
  - \(<a, 15> <b, 15>\)
  - \(<h, 1> <i, 1> <B, 2> <A, 8> <\text{main}, 14>\)
### Symbol Table: Dynamic Scoping

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. }

---

### Symbol Table: Dynamic Scoping

- **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>

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

---

### The Meaning of Names within a Scope

- **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
  - Euclid scope rules are designed to prevent this

---

### C++ Aliasing - Passing by Reference

- 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

---

### Aliasing

- 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
The Meaning of Names within a Scope

• 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
    • Ada
    • C++ (and hence Java and C#)

The Meaning of Names within a Scope

• 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;)
    • polymorphic functions – one thing that works in more than one way
      • code takes a list of types, where the types have some commonality that will be exploited.
      • used in Ada and Smalltalk, primarily

The Meaning of Names within a Scope

• 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);
  no : month;
  pb : print_base;
begin
  no := dec; -- the month dec
  pb := oct; -- the print_base oct
  print(month'('oct'));
  -- must be able to provide context explicitly
  print (month'('oct'));
  -- Modula-3 and C# require enumeration prefixes
  mo := month.dec;
  pb := print_base.oct;

The Meaning of Names within a Scope

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

The Meaning of Names within a Scope

• 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;

The Meaning of Names within a Scope

• 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
The Meaning of Names within a Scope

• polymorphism
  – multiple forms
  – two types
    • parametric polymorphism
    • subtype polymorphism

• 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

The Meaning of Names within a Scope

• explicit parametric polymorphism in Ada generics
  
```ada
  generic
  type T is private;
  with function <(x : y : T) return Boolean;
  function min(x : 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 data_min is new min(data, data_preceded);
```

The Meaning of Names within a Scope

• implicit parametric polymorphism in Scheme
  
```lisp
  define min (lambda (a b) (if (< a b) a b))
```

• implicit parametric polymorphism in Haskell
  
```haskell
  min a b = if a < b then a else b
```

Separate Compilation

• 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

• separately-compiled files in C (continued)
  – `extern` on a variable or function means that it is declared in another source file
  – function headers without bodies are `extern` by default
  – `extern` declarations are interpreted as forward declarations if a later declaration overrides them
Separate Compilation

• 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

• 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