Chapter 3:: Names, Scopes, and Bindings (cont.)

Programming Language Pragmatics

Michael L. Scott

Review

- What is a regular expression ?
- What is a context-free grammar ?
- What is BNF ?
- What is a derivation ?
- What is a parse ?
- What are terminals/non-terminals/start symbol ?
- What is Kleene star and how is it denoted ?
- What is binding ?

Review

- What is early/late binding with respect to OOP and Java in particular ?
- What is scope ?
- What is lifetime?
- What is the "general" basic relationship between compile time/run time and efficiency/flexibility ?
- What is the purpose of scope rules ?
- What is central to recursion ?

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

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

Lifetime and Storage Management

- Heap for dynamic allocation

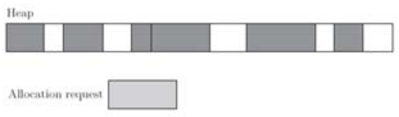


Figure 3.3: **External fragmentation.** The shaded blocks are in use; the clear blocks are free. While there is more than enough total free space remaining to satisfy an allocation request of the illustrated size, no single remaining block is large enough.

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

- 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

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

Scope Rules

- 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
  - These have even more sophisticated (static) scope rules
- Euclid is an example of a language with lexically-nested scopes in which all scopes are closed
  - rules were designed to avoid ALIASES, which complicate optimization and correctness arguments

Scope Rules

- 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
  - You access a variable in a scope k levels out by following k static links and then using the known offset within the frame thus found
- More details in Chapter 8

Scope Rules

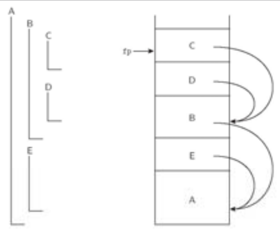


Figure 3.5: **Static chains.** Subroutines A, B, C, D, and E are nested as shown on the left. If the sequence of nested calls at run time is A, E, B, D, and C, then the static links in the stack will look as shown on the right. The code for subroutine C can find local objects at known offsets from the frame pointer. It can find local objects of the surrounding scope, B, by dereferencing its static chain once and then applying an offset. It can find local objects in B's surrounding scope, A, by dereferencing its static chain twice and then applying an offset.

Scope Rules

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

Scope Rules

- 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

```
program scopes (input, output );
var a : integer;
procedure first;
   begin a := 1; end;
procedure second;
   var a : integer;
   begin first; end;
begin
   a := 2; second; write(a);
end.
```

Scope Rules
Example: Static vs. Dynamic

- 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

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

Scope Rules
Example: Static vs. Dynamic

- 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

Binding of Referencing Environments

- 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

- REFERENCING ENVIRONMENT of a statement at run time is the set of active bindings
- A referencing environment corresponds to a collection of scopes that are examined (in order) to find a binding

Binding of Referencing Environments

- 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

Binding within a Scope

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

Binding within a Scope

- Overloading
  - some overloading happens in almost all languages
    - integer + v. real +
    - read and write in Pascal
    - function return in Pascal
  - some languages get into overloading in a big way
    - Ada (see Figure 3.18 for examples)
    - C++ (see Figure 3.19 for examples)

Binding 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;)
complex norm (complex c ) { // ...
```

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

Binding within a Scope

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

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
  - Functions 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

- The morals of the story:
  - language features can be surprisingly subtle
  - designing languages to make life easier for the compiler writer *can* be a GOOD THING
  - most of the languages that are easy to understand are easy to compile, and vice versa

Conclusions

- 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

Chapter 5:: Target Machine Architecture

Programming Language Pragmatics

Michael L. Scott

Assembly-Level View

- As mentioned early in this course, a compiler is simply a translator
  - It translates programs written in one language into programs written in another language
    - This other language can be almost anything
    - Most of the time, however, it's the machine language for some available computer

Assembly-Level View

- As a review, we will go over some of the material most relevant to language implementation, so that we can better understand
  - what the compiler has to do to your program
  - why certain things are fast and others slow
  - why certain things are easy to compile and others aren't

Assembly-Level View

- There are many different programming languages and there are many different machine languages
  - Machine languages show considerably less diversity than programming languages
  - Traditionally, each machine language corresponds to a different computer ARCHITECTURE
  - The IMPLEMENTATION is how the architecture is realized in hardware

Assembly-Level View

- Formally, an architecture is the interface to the hardware
  - what it looks like to a user writing programs on the bare machine.
- In the last 20 years, the line between these has blurred to the point of disappearing
  - compilers have to know a LOT about the implementation to do a decent job

Assembly-Level View

- Changes in hardware technology (e.g., how many transistors can you fit on one chip?) have made new implementation techniques possible
  - the architecture was also modified
  - *Example*: RISC (reduced instruction set computer) revolution ~20 years ago
- In the discussion below, we will focus on modern RISC architectures, with a limited amount of coverage of their predecessors, the CISC architectures

Workstation Macro-Architecture

- Most modern computers consist of a collection of DEVICES that talk to each other over a BUS
- From the point of view of language implementation:
  - the most important device is the PROCESSOR(S)
  - the second most important is main memory
  - other devices include: disks, keyboards, screens, networks, general-purpose serial/parallel ports, etc.

Workstation Macro-Architecture

- Almost all modern computers use the (von Neumann) stored program concept:
  - a program is simply a collection of bits in memory that the computer *interprets* as instructions, rather than as integers, floating point numbers, or some other sort of data
- What a processor does is repeatedly
  - fetch an instruction from memory
  - decode it - figure out what it says to do
  - fetch any needed operands from registers or memory
  - execute the operation, and
  - store any result(s) back into registers or memory

Workstation Macro-Architecture

- This set of operations is referred to as the fetch-execute cycle
  - The computer runs this cycle at a furious pace, never stopping, regardless of the meaning of the instructions
    - You can point a processor's instruction fetch logic at a long string of floating point numbers and it will blithely begin to execute them; it will do *something*, though that something probably won't make much sense
    - Operating systems are designed so that when the computer has nothing useful to do it is pointed at an infinite loop that it can execute furiously, but harmlessly

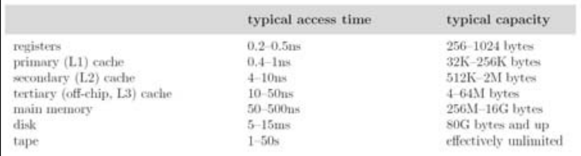
Workstation Macro-Architecture

- The crucial components of a typical processor include a collection of FUNCTIONAL UNITS:
  - hardware to decode instructions and drive the other functional units
  - hardware to fetch instructions and data from memory and to store them back again if they are modified
  - one or more arithmetic/logic units (ALUs) to do actual computations
  - registers to hold the most heavily-used parts of the state of the computation
  - hardware to move data among the various functional units and registers

Memory Hierarchy

- Memory is too big to fit on one chip with a processor
  - Because memory is off-chip (in fact, on the other side of the bus), getting at it is much slower than getting at things on-chip
  - Most computers therefore employ a MEMORY HIERARCHY, in which things that are used more often are kept close at hand

Memory Hierarchy



	typical access time	typical capacity
registers	0.2–0.5ns	256–1024 bytes
primary (L1) cache	0.4–1ns	32K–256K bytes
secondary (L2) cache	4–10ns	512K–2M bytes
tertiary (off-chip, L3) cache	10–50ns	4–64M bytes
main memory	50–500ns	256M–16G bytes
disk	5–15ms	80G bytes and up
tape	1–50s	effectively unlimited

Figure 5.1: **The memory hierarchy of a workstation-class computer.** Access times and capacities are approximate, based on 2005 technology. Registers must be accessed within a single clock cycle. Primary cache typically responds in 1–2 cycles; off-chip cache in more like 20 cycles. Main memory on a supercomputer can be as fast as off-chip cache; on a workstation it is typically much slower. Disk and tape times are constrained by the movement of physical parts.