Chapter 3 - Names, Scopes, and Bindings

May 23, 2016

Origins of high level languages

- With machine languages, generally there are no names, limited scoping, and binding is not really a relevant concept.
- All assemblers allow some sort of names; the most minimal ones supported are generally called labels and definitions, but fancier assemblers can have other types of names such as macros.

Origins of high level languages

- Scope comes into play even in assemblers. The most obvious of course is between separate compilation modules; however, even within an assembler, the idea of local labels (often indicated with an initial "L", "%%", or perhaps ".") is very convenient — think about how macros work, with simple text rewriting of one string for another with designated abstraction points. If you have a label created by a macro, what happens if you invoke the same macro again in the same module? Unless you make it part of the parameterization, it will be the same label. If you have the ability to declare very local labels, though, this isn't a problem.

Origins of high level languages

- Since assembly languages actually provide the greatest freedom (all of the machine's resources are there for the programmer in their fullest extent), what is the rationale for "higher level" languages?
- Obviously, you get more (if not perfect) machine independence. No longer do you care about the *most* that you can do with the hardware; it becomes instead using the greatest *common* hardware efficiently (think gcd() rather than max()).
- In assembly, even with heavy use of macros, porting code from one architecture to a reasonably differing one requires more work than any high level language.

Origins of high level languages

- More imporatantly, higher level languages are generally good at something; specialized languages, like gp/pari, are even "superb" rather than "generally good".
- While you lose a certain amount of freedom, you (should) gain a lot of *expressivity*, the ability to pithily and *clearly* express ideas.
- For instance, look at the expressivity of Haskell for the "hello_world" program of the functional world, the factorial function.

So what is a name?

- Your text gives this definition on pages 111-112:
  "A name is a mnemonic character string used to represent something else. Names in the most languages are identifiers (alphanumeric tokens), though certain other symbols, such as $+$ or $:=$, can also be names. Names allow us to refer to variables, constants, operations, types, and so on using symbolic identifiers rather than low-level concepts like addresses."

So what is a name?

- Names also allow us to *abstract*; that is, use a name to represent a slew of activities.

3.1 The notions of binding and binding time

- A “binding” is an association between a name and what it represents.
- “Binding time” is the time when we decide on an association.

3.1 Possible binding times

- Language design time (control flow, constructors, fundamental types; nice recent example
- Language implementation time (precision, i/o, stacks, heaps, other memory blobs, hardware exceptions)
- Program writing time
- Compile time (sections; debugging information; any static data has to fit somewhere)

3.1 Possible binding times

- Link time (resolution of "static" intermodule references; any mechanics needed to deal with dynamic loading)
- Load time (more dynamic loading issues; resolving sectioning directives into actual memory layout)
- Run time ("a host of other decisions that vary from language to language" is how your book expresses this on page 113.)

3.1 Binding time

- Earlier binding generally implies more efficient execution of the final code; later binding of names generally implies slower code.
- Generally, “static” is used to mean “before run time”; “dynamic” is used to mean “at run time”. (Not exactly precise there; how about when dynamic linking is happening, for instance? Static? Dynamic?)

3.2.1 Object life and storage management

- Static allocation: obviously, global variables can be statically allocated. The text argues on page 115 that you can regard the machine language instructions of a compiled program as being statically allocated. Fixed strings are statically allocated. Debugging structures that support debuggers like gdb are statically allocated (see the dwarf2 specification, for instance.)

3.2.1 Object life and storage management

- Static allocation: Even local variables might be allocated statically; the canonical example would be older Fortran subroutines, which could not be recursive. Named constants, even when local to a subroutine, might well be statically allocated if there the language and the code allow sufficient clarity as to the "constant's" value.

3.2.1 Object life and storage management

- Arguments and return values: One of my favorite sentences from the text is on page 116: "Modern compilers keep these in registers whenever possible, but sometimes space in memory is needed." Now, if we could get more operating systems to use the same regimen for system calls! (Linux, to its credit, does do so; the BSDs, as illustrated above, don't.)
- Temporaries: (obviously) use registers whenever possible.

3.2.2 Stack-based allocation

- Subroutine recursion means that we must have an ability for dynamically allocating the local state for a subroutine (if only one instance of a subroutine can exist (as is the case for older Fortran), then we can use static allocation for the subroutines local state.)

3.2.2 Stack-based allocation

- Most languages use a stack of activation records to allocate local state. However, not all do so: look at this discussion about adding such a facility to Forth (indeed, such allocation was added to some Forth dialects). In the more modern age, Haskell also does not: see this discussion of how to do tracing. (If you want to search for more examples, use search terms something like "stackless programming languages".)

3.2.2 Stack-based allocation

- Who maintains the stack? The caller, with the "calling sequence", and the callee, with the prologue and the epilogue.
- Generally, the activation record's structure is statically determined and the compiler can use fixed offsets from a stack frame pointer into the activation record.
- Most machine languages include a push/pop in the instruction set, and sometimes these can be quite sophisticated, pushing many registers for instance simultaneously onto the stack.

3.2.2 Stack-based allocation

- While stack-based allocation can provide some small savings even in non-recursive programming languages (viz., pages 118-119), stack-based allocation is a widely exploited mechanism with respect to security.

3.2.3 Heap-based allocation

- The first thing to understand is that a program is not *required* to have exactly one heap. Assembly language programs often have no heap (i.e., in Unix/Linux, no calls to sbrk(2) are made); it is also possible to have multiple heaps (see malloc(3) man page on using mmap(2) to create separate and memory blocks from the sbrk(2) maintained heap.) In OpenBSD malloc, only mmap(2)-based structures are used; the allocator Hoard uses both sbrk(2)-based and mmap(2)-based allocation.

3.2.3 Heap-based allocation

- Heap management is a subject of long and intense study. There are lot of ideas; my own opinion is that OpenBSD/Hoard approaches are far better than trying to do just single contiguous heap approaches, and I think that the research shows that not only is this faster and more secure, it's also conceptually simpler.

3.2.4 Garbage collection

- Some languages assume implicit deallocation when there are no remaining references to an object (Perl, for instance). This assumed (and thus automatic) deallocation has been termed "garbage collection."

3.2.4 Garbage collection

- Automatic garbage collection saves the programmer from many possible errors in manual deallocation: dangling references, use after free(3)-ing a reference, and double free(3)s are all quite common problems with manual schemes.

3.3.1 Scope rules: Static scoping

- Static (aka "lexical") scoping means that the bindings between names and objects can be determined at compile time without any reference of the flow of control at run time. The most common variation is for the latest declaration to be the current binding for a name.

3.3.1 Scope rules: Static scoping

- Nested scoping rules: in languages which allow nested routines, it is very common for a variable name from a parent routine to be available to a child routine.
- How does a child does this? With stack-based languages, generally a child activation keeps a pointer to its parent activation (not necessarily the same lexical parent/child relationship, though, which could be grandparent/child relationship, for instance.)

3.3.3 Declaration order

- Some languages require "declarations first, code after", but still what happens if a refers to b and b refers to a? Or what about the classic Pascal conundrum on page 128?

3.3.3 Declaration order

- Sort of a solution: forward declarations
- Good idea also: order of declarations doesn't matter; it's in scope in the entire lexical block.
- Another idea: have no variable declarations (typical of many scripting languages)
- Another idea: Like Scheme, support all reasonable versions of declaration order with let, letrec, and let* (even Perl isn't this flexible!)

3.3.3 Declaration order

- How do you have mutual recursion, where A calls B calls A? One has to be defined before the other, and thus there is a dilemma.
- You could take the C route, of having "declarations" and "definitions" be different things; then you could declare both A and B, and then both function definitions would be aware of the other.

3.3.3 Declaration order

- Unnamed code blocks: many languages allow you to have a block, with full scope rules, wherever a statement can occur. This is quite handy when you need a temporary variable that can be allocated on the stack rather than having to create it on a heap.

3.3.3 Declaration order

- Some languages, like ML and Forth, allow you to redefine a name, but also keep the previous behavior for definitions that use the original behavior.

3.3.4 Modules

- This methodology became common in the late 70s and early 80s. The idea was to encapsulate names, and control visibility. Concepts like “import”, “export”, and “hiding” were brought into play. (The book isn’t quite precise about Perl, by the by; Perl does indeed use “package” as a keyword to indicate a module, but the whole of a file named “x.pm” and the “package x” inside the file is called a module in Perl-speak.)

3.3.4 Modules: closed and open scope

- Modules into which names must be explicitly imported are "closed scopes."
- Modules that do not not require imports are "open scopes".
- "Selectively open" allows the programmer to add a qualifier; for instance, if A exports buzz, then B can automatically reference it as A.buzz; it becomes visible if B explicitly imports the name.

3.3.5 Module types and classes

- Some languages treat modules as a mediator, managing objects with various means to express the semantics destroy/create/change
- Some languages treat modules as defining “types”, rather than a mere mechanism for encapsulation.

3.3.5 Module types and classes

- Some languages extend the “module as types” concept to “modules as a class”, with the extension of the idea of “inheritance”, allowing the new class to “inherit” characteristics of a parent class.

3.3.6 Dynamic scoping

- When the bindings between names and objects depend on the flow of control at run time rather than just lexical scope, you have “dynamic scoping”.
- TeX, for instance, uses dynamic scoping, as does Forth. Perl allows for both, as noted in footnote 10 at the bottom of page 140.

3.4 Implementing scope

- To keep track of names in a statically scoped language, the *compiler* for that language merely keeps a symbol table, mapping a name to the information known about the symbol.
- To track names in a dynamically scoped language, the *run-time* for that language must keep up with the mapping between names and objects.

3.5 The meaning of names within a scope

- Aliases (yipes!?): aliasing (i.e., two different names in a scope that refer to the same object) makes it harder to understand, much less optimize, programs.
- Overloading: the trick with overloading of operator and function names is usually being able to distinguish the versions by the types of the arguments.

3.5 The meaning of names within a scope

- Redefining built-ins: many languages allow much freedom in redefining built-ins; Perl, for instance, has some very sophisticated methods, like TIE, which can redefine even the very definition of a built-in type.

3.5.3 Polymorphism, coercion, and related

- If a function expects some an argument of type A, but finds one of type B, it might try to "coerce" the argument to type B. This type of implicit coercion (one of the fundamental building blocks of Perl), is somewhat controversial. C and C-like languages generally allow for this only in very limited circumstances, such as converting integers silently into reals.

3.5.3 Polymorphism, coercion, and related

- Polymorphism allows a function to abstract out the idea of type; generally, this is referred to by ideas like “genericity”; Ada is perhaps the best-known for its generics; C++ has its templating. Generics and templates are generally handled by having separate code for each instantiated type.

3.6 Binding of referencing environments

- Generally, static scoping is combined with deep binding; however, dynamic scoping often entails mechanisms for expressing both shallow binding and a deep binding (generally with "closures").

Closures

- "Deep binding is implemented by creating an explicit representation of a referencing environment (generally the one in which the subroutine would execute if called at the present time) and bundling it together with a reference to the subroutine." This bundle can be referred to as a *closure*, and can be passed to *other* routines for later execution.
- Some languages, such as C, don't have any formal mechanism for expressing closures, but can simulate closures to a degree by using function pointers and callbacks that reference explicit context state (often abbreviated by "ctx_").

3.6.2 First-class values and unlimited extent

- First class value: it can be passed as a parameter, returned from a subroutine, or assigned into a variable.
- Second class value: it can be passed as a parameter. It cannot be returned from a subroutine or assigned into a variable.
- Third class value: it cannot be passed as a parameter, returned from a subroutine, or assigned into a variable.

3.6.2 First-class values and unlimited extent

- Typically, integers, e.g., are first class values, and labels are third class values.
- Subroutines in functional langauges are first class; usually first class in most scripting languages, and often second class in most other languages.

3.7 Macro expansion

- As we saw earler, more sophisticated assemblers make use of macros; C preserved that idea. A macro is an straight text expansion

```
#define emit(a)  printf("%d",a)
```

- Becomes via the magic of “gcc -E”

```
emit(3); ==> printf("%d",3);
```

- Macros were often used to create something like a template/generics facility