Dynamic Scoping

What is DynamicScoping and LexicalScoping?

Scoping itself is how you search for a variable with a given name. A variable has a *scope* which is the whole area in which that variable can be accessed by name. If there is a reference to a variable "a" then how does the compiler or interpreter find it?

In lexical scoping (and if you're the interpreter), you search in the local function (the function which is running now), then you search in the function (or scope) in which that function was *defined*, then you search in the function (scope) in which *that* function was defined, and so forth. "Lexical" here refers to *text*, in that you can find out what variable is being referred to by looking at the nesting of scopes in the program text.

In *dynamic* scoping, by contrast, you search in the local function first, then you search in the function that *called* the local function, then you search in the function that called *that* function, and so on, up the call stack. "Dynamic" refers to *change*, in that the call stack can be different every time a given function is called, and so the function might hit different variables depending on where it is called from.

Dynamic scoping is useful as a substitute for globally scoped variables. A function can say "let current_numeric_base = 16; call other functions;" and the other functions will all print in hexadecimal. Then when they return, and the base-setting function returns, the base will return to whatever it was.

It can also have a use similar to a ContextObject, as in JavaServlets and EJB APIs, holding state for a series of subroutines.

The two most popular methods for implementing DynamicScoping are DeepBinding and ShallowBinding.

See GuySteele and GeraldSussman's paper TheArtOfTheInterpreter for discussion of static vs dynamic scoping and more.

[The following is somewhat specific to LispSchemeDifferences and the use of dynamic variables in CommonLisp.]

What does dynamic scoping do for *standard-output* that a temporary assignment to a global does not? I presume that it would be simple enough to write a macro that saves the old value, assigns a new one, executes a form, and restores the old value before returning the value of the form? Wouldn't such a macro simply be implementing dynamic scoping in terms of lexical scoping?

GuySteele's original papers on the SchemeLanguage papers showed that dynamically scoped variables are equivalent to the controlled use of global variables. So yes, writing a macro to save, change and restore arbitrary global variables actually implements dynamic scoping. Global variables don't really have lexical scope, however, so it's not "in terms of" lexical scoping. Unfortunately, dynamic scoping (along with global variables) begins to break down in multi-threaded situations.

*If you just save the variable, assign to it, and restore it, it won't be thread safe, and it will require a special binding construct. Real SpecialVariables work with any binding construct: LET, MULTIPLE-VALUE-BIND, WITH-OPEN-FILE, etc; the special nature of the binding is remembered as a property of the symbol and applied accordingly. And in Lisp implementations that support threads, the bindings are thread-specific! Implementing thread-specific storage is not trivial. If you assign to *standard-output*, you have a race condition. Assignment is not binding.*

I agree with AlainPicard that sometimes dynamic scope is what you want. (Though I don't want it as often as he does, since global *PRINT-BASE* in particular has caused me problems. I wish the print base were an attribute of streams, instead of a global variable.) But SunirShah's original proposal [on LispSchemeDifferences] was to use dynamic scoping to replace all lexical scoping. That would be a very bad idea, most of all because it would replace the clean semantics of closures with semantics so twisted that closures couldn't be used in non-trivial code without creating horrible maintenance problems. -- BillNewman

*Bill, *PRINT-BASE* et al are **special**, not **global**. There is a world of difference. -- ap*

Yeah. I think what Bill is talking about, though (and he can refactor his words and delete my comment if I'm wrong) is that *print-base* et al apply to output on *every* stream, instead of being individually settable on each. For example, I have an application which uses the printer to serialize data into a database: suppose that the debugger binds *print-length* to something non-nil for human readability, then the programmer calls my SAVE-OBJECT function on a list from the break loop, so it saves something which will later be unREADable. Basically, SAVE-OBJECT has to be very careful to rebind all the printer-related variables to known-good values.

This is exactly why your application should re-bind the required variable at a low point in the call stack, rather than depend on what may accidentally be in existence. The WITH-STANDARD-IO-SYNTAX is provided specifically for this purpose. -- ap

---

I find that dynamic scoping makes spaghetti code out of modular code. Everything depends on order of execution, and it quickly becomes impossible to understand what's going on without a debugger open. Just an opinion, however... --Anthony

Lexical scoping is easier to program with, and is necessary for safe closures, callbacks, CallWithCurrentContinuation, HigherOrderFunctions etc.

*Consider variables like *standard-output*, *print-base*, *break-on-signals*, etc... Dynamic scoping has the* potentiality *of turning code into a mess, but, hey, all powerful constructs are like that. It's certainly a much less dangerous feature than, say, call-cc, which is so powerful it prevents a reasonable implementation of UNWIND-PROTECT. -- AlainPicard*

Hmm. I was under the impression that 'dynamic-wind' solved these issues, although it is a fairly recent addition if I'm not mistaken. -- WilliamUnderwood

---

Richard Stallman makes a good case for dynamic scoping in his paper on emacs: http://www.gnu.org/software/emacs/emacs-paper.html#SEC17

Incidentally, emacs ships with macros that support lexical scoping in emacs lisp (emacs lisp variabels are dynamically scoped by default, though lexical scoping turned on on a per-file basis is supported since version 24) This would suggest that dynamic scoping is the more general of the two - or is it possible to implement dynamic scoping in a lexically scoped language in a similar way?

Possibly. A trivial, but non-threasafe, implementation follows in pseudocode (because I don't know lisp) which is based on lisp and perl six and should probably be rewritten. Also, since Assembly has neither, yet is used to implement both several times, your argument may be irrelevant.

```
     define-macro-variable dynamic-names = []
     defmacro "defdyn" (a : Name, (optional "=", value = NULL))
      defvar code = q:code{
         defvar #{a}-dynamic-list = ["value"];
      }
      dynamic-names += a
      (type-of a) = Dynamic[(err (type-of value) (infer-type-of-
slot a))]
      return code
    defmacro [X] (a : Slot[Dynamic[X]])
      return {(head #{a}-dynamic-list)}
    defmacro "enter-dynamic-scope"
      return (reduce dynamic-names q:code{} (lambda (a : Code, b
: Name)
        a + q:code { #{b}-dynamic-list = (cons (head #{b}-
dynamic-list) #{b}-dynamic-list);}))
    defmacro "leave-dynamic-scope"
      return (reduce dynamic-names q:code{} (lambda (a : Code, b
: Name)
        a + q:code { #{b}-dynamic-list = (tail #{b}-dynamic-
list)}))
```

Moved to DynamicScopingInSmalltalk

See DynamicClosure, LexicalScoping, ScopingRules, ScopeAndClosures, SpecialVariable

---

CategoryCodingConventions

Last edit June 24, 2013, See github about remodeling.