Dynamic scope in Lisp
Supplementary notes for lecture 23, CS 421 (spring, 2010)
S. Kamin

This note fills in a topic that was mentioned in class but not explained. Specifically, I mentioned it when discussing the last slide in lecture 23. Because it wasn't covered in class, you will not be responsible for it on the exam. But, aside from historical interest, understanding dynamic scope will help you better understand static scope, so I'm providing this explanation which I hope you will read.

Static, or *lexical*, *scope* is most natural and familiar to all programmers. Static scope just means that, when you see a name being used in a program, you can point to the corresponding declaration of that name: the correspondence, or binding, is determined by the text of the program. For the most part, in common programming languages, the use of a name is bound to the *closest enclosing* declaration of that name.

For example, in Java, if a method has a local variable named x and a field named x, uses of x inside the method are bound to the local declaration. In Ocaml, if there is a function called f defined at the top level, but an expression defines and uses f locally – as in "let f x = … in … f …" – then the local use of f is bound to the local definition.

Finding the declaration of a name may involve looking in different source files – for example, in OCaml, you have to look at included modules, and in Java, you have to look at other files in the current and imported packages. But as long as the source code is available, you can in principle point to a specific declaration of each name.

There is a common exception to this rule in object-oriented languages, and this is the one that is mentioned in the notes. *Dynamic binding* is used for virtual functions (those declared virtual in C++, or any method in Java). Because of inheritance, a method call "x.m()" may refer to one of several methods named m. If you have all of the program text, you can determine the set of all methods that might be bound to this use, although a programmer can always add to that list by adding a new subclass and overriding m. But you cannot, in general, point to *one* declaration of m that is bound to this use. This form of dynamic binding leads to difficulties in understanding and reasoning about programs, but it lies at the very heart of object-oriented programming; it was what allows for significant code re-use. Therefore, it is generally considered a Good Thing.

The scope rule in the original version of Lisp was dynamic, in a different way, and that is what I want to explain in this note. Although this form of dynamic binding does allow for some helpful uses, it is widely considered to be a mistake. John McCarthy has acknowledged that it was introduced somewhat unconsciously; it is the form of binding that you get by using an obvious, but incorrect, implementation of user-defined functions.

The binding rule in Lisp is this: *a use of a name is bound to the <u>most recent</u> declaration of that name that is still live*.

For a declaration to be live normally means that the function that includes that declaration has not finished executing. This notion corresponds to the usual, stack-like function call and return. For this reason, the Lisp dynamic scope rule is normally the same as static scope.

However, higher-order functions throw a wrench in the works. Consider this function (using OCaml syntax):

```
let add = fun x -> fun y -> x+y
```

and this application:

```
let add3 = add  3
```

If we think of add3 as having value "fun y -> x+y", then the question becomes this: at the moment add3 is applied – which may be much later – where will it get the value of x?

Here, the answer given by Lisp is different from Ocaml's. In OCaml, x will be 3 *whenever* add3 is applied – this works in both the substitution and closure models, as well, of course, as the implementation of OCaml. The scope rule of OCaml says that the original declaration of x in the definition of add, which is bound to 3, is the one used by the x in the body of add.

In Lisp, however, it depends on whether there is a more recent, live declaration of x. Specifically, consider this case:

```
let add = fun x -> fun y -> x+y;;
let add3 = add 3;;
let f x = add3 x;;
f 4;;
```

Under dynamic binding: When add3 is applied in the body of f, there is a live declaration of x – the parameter of f – that is bound to x. It has value 4. Since this is the *most recent* binding of x at the time add3 is applied, and since it is still live when add3 is applied, it is the binding used inside add3. Thus, f 4 returns 8. (Under static binding, f 5 will of course return 7.)

Here is another example:

```
let h f = let x = 3 in f x;;
let f x = let g y = x + y in h g;;
f 5;;
```

Under static binding, in the call f 5, g becomes "fun y -> 5+y". This is passed to h, so the call "h g" becomes "let x=3 in (fun y -> 5+y) x", which is the same as "(fun y -> 5+y) 3", which evaluates to 8. Under dynamic binding, g is just "fun y -> x+y", and the call "h g" becomes "let x=3 in (fun y -> x+y) x". Both values of x are 3 and the value of the expression is 6.

The short version of how this happened is that the Lisp implementers didn't realize that it was necessary to build closures.

The problem with dynamic scope here is that it renders the action of higher-order functions like add and h completely unpredictable. If a higher-order function is used in a context where some function with a local variable with the wrong name happens to be on the stack, the higher-order function will behave differently. Because of this unpredictability, higher-order functions become nearly useless. And because higher-order functions are potentially very useful, the use of dynamic scope in Lisp is considered an error in the language design.