Alan Moody

2/9/16

**Efficiency**:

Any stack-based implementation could run up to O(n^2). Example:

interpret(expr){

if thing, thing

if thing, thing

if ID, subst-linear(ID, expr)

}

subst-linear(ID, expr){

for(binding : expr){

if ID = binding.ID, return binding.value

}

}

Here our subst-linear function has a complexity of O(n). If we're substituting all of the IDs

in the entire expression, we'll "iterate" through them as we do the recursion and end up with

O(n^2) because we have to do the linear search every time.

We decided to use a binary search tree, containing bindings stored as ID-value pairs,

sorted alphabetically by ID.

Pseudocode:

interpret(expr){

if thing, thing

if thing, thing

if ID, subst-BST(ID, expr)

}

subst-BST(ID, expr){

return binary\_search(ID, expr.bindings).value

}

Binary search has a complexity of O(log n) in the average case. If the bindings are put in

to the tree in order, then we can have O(n) complexity and no improvment over the linked-list

setup. However, in a situation where O(log n) is going to be noticeably different from O(n),

this is unlikely. We'll have to substitute each ID that appears in the expression, so we could

end up with something like O(n log n) as we "iterate" through all the IDs in the expression.

**Bitdiddle Algorithm**:

Ben is wrong in general. Try:

{with {x 3}}

{with {x 4}

{with {f {fun {y} {+ x y}}}

{with {x 5}

{f 10}}}}

This will evaluate to 13 with Ben's method of dynamic scoping. Obviously, however, this is wrong;

x has been set to 3, but should have been overridden by the {with {x 4}...} in the next expression.

In effect, Ben wants to set global variables that all end up being static variables. The oldest

value of some variable in the environment will never change if we always ignore later assignments,

so we have what are now static variables. In the single example Ben is looking at, the result

happens to be the same, but Ben is generally wrong because he's forgotten that a variable is

supposed to vary, and that our considerations of scope need to keep in mind the location of a

variable relative to the current expression, not necessarily its age.

**Application**:

First-class functions "feel" right. Aside from the practical implications of not having to use

a table full of pointers to do any fun stuff with functions, it's a great way to think about

what computers are supposed to accomplish. Ideas are principles of action, and so it makes sense

to consider functions as the embodiment of ideas, and let them interact in that way. As much as

I'd like to do a lot of programming in a Lisp-like environment going forward, that seems unlikely.

However, having learned about first-class functions, even in languages that don't have them, I

can remember what I've learned here and design functions that are independent from each other

and therefore write higher-quality code with less redundancy and entanglement.