2013W1-lecture13

October 20, 2013

Contents

1 Question of the Day

OK, this one isn't really a question. I admit it!

Why should I care about Programming Languages? The spam botnet administrator edition: http://www.youtube.com/watch?v=ILOtIMShi9s, roughly 33:00.

This is also a tremendously fun video of a talk by a grad school friend of mine on high highly influential research into the black market world of computing.

2 Finishing those old notes

file:2013W1-lecture7.org

3 Environments and Scope

Substitution didn't really capture the typical intuition people have for how evaluation of identifiers works. We usually imagine "looking up the value" of the identifier in memory or in some sort of data structure.

Let's take a few steps toward that model.

3.1 Scope: Static vs. Dynamic

First, let's figure out where an identifier **should** be available.

The region of a program where an identifier binding is available is called the identifier's *scope*.

Historically, there's two main kinds of scope. *Static scope* follows the static (textual, lexical, or "on the page") form of the program. *Dynamic scope* follows the function call structure of the program.

Consider our `use-x/define-x` example from our function test cases:

```
(define (use-x y)
  x)

(define (define-x x)
  (use-x 5))
```

Under static scoping:

- Where is the binding for `y` available?
- Where is the binding for `x` available?
- What happens when we call `(define-x 3)`?

Under dynamic scoping:

- Where is the binding for `y` available?
- Where is the binding for `x` available?
- What happens when we call `(define-x 3)`?

Here's an even more interesting pair of questions. Be careful, the answers are not both obvious from what we said above.

- Under static scoping, where does the binding for `x` **come from**?
- Under dynamic scoping, where does the binding for `x` **come from**?

Which one is better: static or dynamic scoping?
Let's go play around with scope in file:2013W1-environment-diagrams.rkt.
Does our implementation of `subst` do the right thing for scoping?

3.2 Environment

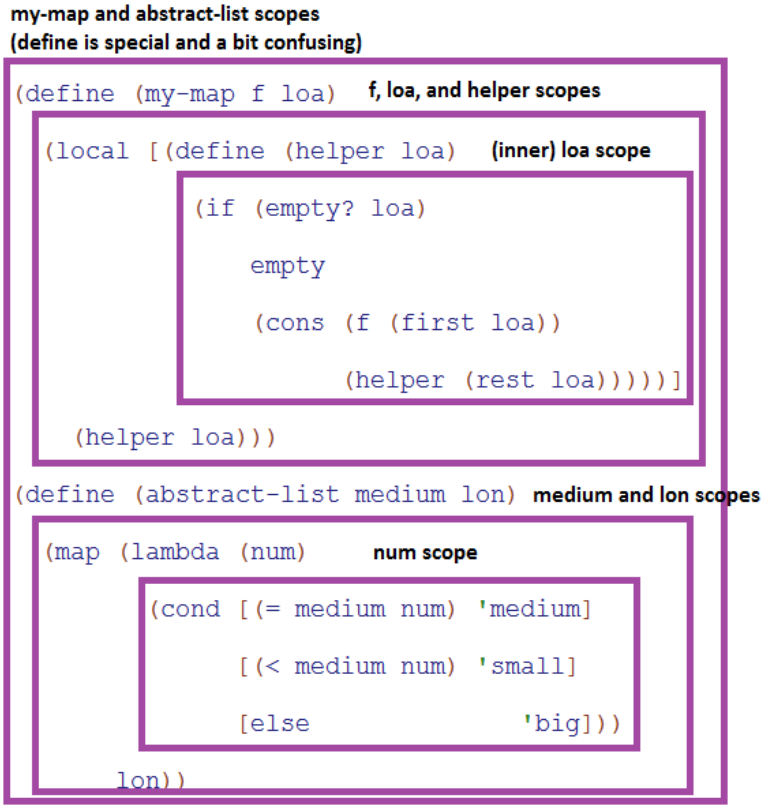
Now that we know where identifier bindings are available, we need a data structure in which to look up identifier references. We'll call that data structure an *environment* because it defines what's "around us" at any given point as our program runs.

Because we decided on static scoping, there's something nifty about our environment data structure. We know **exactly** which identifiers will be bound in the environment at every point in our program before our program even starts running, but we do **not** know what values they'll be bound to.

Check out the `my-map` function in file:2013W1-environment-diagrams:

- Is there one particular value that the `loa` inside `helper` is bound to?
- What about just within a **single** call to `my-map`?

So, our environment follows the shape of the boxes in our environment di-



agram:

There are **many** choices for data structures for environments. Because we know our environment's shape statically, we can even compile away all our identifiers and replace them with direct memory references or indexes into an array!

To keep things clear and simple, we'll just use lists. Specifically: lists of pairs of identifiers and their values. They're **not** the most efficient solution, but they're easy to use, understand, and manipulate in Racket.

3.2.1 Important Side Note: Assignment vs. Binding

Later our environments will pair identifiers with their locations in memory—which we'll call the "store"—rather than with their values. That's because of assignment, which can change the **value** associated with an identifier, but does **not** bind the variable to a new **location**.

The takeaway is that assignment is **not** the same thing as binding. It doesn't create a new identifier; it just changes the value associated with an

existing identifier.

In a less important side note: some languages—particularly many "scripting languages" like Python—use a syntactic assignment statement that serves both the purpose of creating a binding and assigning to an variable's value.

So, in Python, we could say:

```
def some_function():
  y = 2
  x = y + 5
  return x
```

3.3 Implemented Environments

You've already seen environments implemented in the textbook. So, instead, let's break our test cases!

We'll do that next set of notes!

4 Logistics

4.1 Programming Assignment #3

Milestone due today! How are you doing?

Final submission due one week from today.

4.2 Midterm #1 on Tuesday

4.2.1 Check out the pre-posted materials

In the "notes" section of the webpage. This goes beyond (more than the paper from the class exercise) and the questions from Wednesday's notes. Those are *not* the questions from the exam.. but they're sure designed to prepare you for the exam!

4.2.2 Limited (a bit) open notes/open book exam

"Exams will be limited open-book/open-notes: During the exam you may bring for your own use up to 3 textbooks (of your choice, although we recommend CS books) and one 3-ring binder of paper or its equivalent (about 3 inches of paper). No calculators, phones, or other electronic equipment is allowed!"

If you choose to bring fewer than 3 textbooks, you can bring some extra paper. Say 1-2 inches per textbook.

If you would prefer this was more Canadian, you can limit yourself to centimeters, but they're shorter than inches, and who sells a 3-cm 3-ring binder?!

4.2.3 Check out the previous exams

On Piazza, **BUT** be aware that these may not make much sense because of changes to the course structure and textbook.

4.2.4 No appendix provided for this midterm :(

There's been no activity on the wiki.. but feel free to make one for the next exam and gain bonus points for contributing. See http://wiki.ubc.ca/Course:CPSC311/2013WT1

4.3 No quiz on Wednesday

The midterm will do.

5 What have we learned today?

- From QotD: Yet another place where a little programming language might come in handy in an unsavory career path that is hopefully not in your future.
- Scope
  - Define the terms *scope*, *static scope*, and *dynamic scope*.
  - Distinguish between identifier (or variable) *bindings* and assignment.
  - Trace programs using static scope.
  - Trace programs using dynamic scope.
  - Justify static scope as the better option over dynamic scope for software engineering reasons.
  - Sketch the static scopes of identifier bindings in a program.
- Environments
  - Define the term *environment*.
  - Distinguish between static scope of *identifiers* in environments and dynamic extent and identity of their *values*.

    * (That is, we know statically which identifiers will appear where in our environment—and so could do something far more efficient than a list!—but we do not know statically what particular values our identifiers will be bound to and, indeed, those values can "escape" their static context and wind up at distant points in the program. In fact, they'd better be able to... otherwise, how would functions return result values to their callers?!)

```
 ORGMODECONFIG
#+DRAWERS: SOLUTION ORGMODECONFIG
#+COMMENT TODO: change to d:nil (or delete) to not export SOLUTION drawers
#+OPTIONS:   d:t
```