2013W1-lecture14

October 20, 2013

Contents

1 Question of the Day

What is the purpose of the following `bash` syntax? Why bother?

```
echo "${OLDPWD?oops}"

 SOLUTION
$ introduces parameter substitution---usually replacing a reference to
```

```
an environment variable with its value.  The {...} delineate the
expression. OLDPWD is the environment variable's name.

The question mark means "if the environment variable is set, use it,
as though this expression were just ${OLDPWD}; otherwise, print "oops"
and abort the current process with an exit status indicating an error,
specifically 1".

Why bother?

Environment variables are (effectively) supplied by whoever invokes
this process.  They are DYNAMICALLY scoped.  That means we have no
real clue except convention which environment variables are defined
and which are not!

This can be very powerful.. and can cause much trouble.  See, e.g.:
http://peterlyons.com/problog/2010/02/environment-variables-considered-harmful
and
http://www.insectnation.org/howto/environment-variables-considered-harmful

(Thanks for the links, Wikipedia!)
```

2 Logistics

2.1 Midterm #1: Tuesday (tomorrow!) 7PM in CHBE 101

2.2 No quiz on Wednesday

2.3 Programming Assignment #3

Final submission due Friday

2.4 Neat PL research talk upcoming: 24 Oct, 3:30-4:30PM, X836

One bonus point for attending and posting a brief summary of the talk from the CPSC 311 perspective.

Two bonus points for attending and posting a thoughtful, thorough discussion of the talk.

Three bonus points for being the speaker. :)

TypeScript is a programming language whose goal is to support development of large JavaScript programs. TypeScript is a superset of the current JavaScript standard (ECMAScript 5) that adds an optional static type system to JavaScript. TypeScript exists only to support high-level thinking about JavaScript programs; it has no impact on runtime behavior. Because of this, TypeScript is an example of "types for tooling" vs. the more traditional idea of "types for runtime safety." TypeScript has a novel design for type inference; the goal of the design is to provide maximum convenience (few annotations required) and transparency (chains of inference are clear and local). The TypeScript compiler, incremental static analysis tools, and specification are open source (see typescriptlang.org). Several million lines of TypeScript are part of shipping Microsoft products. Since the community preview release in October, 2012, several 100K+ line TypeScript projects have grown up outside of Microsoft and the TypeScript community has created a site, at github.com, that holds over 100 community-maintained TypeScript descriptions of popular JavaScript frameworks such as jQuery.

Steve Lucco is a Technical Fellow at Microsoft, where he is responsible for Microsoft's web development tools and runtimes. He led the development of Microsoft's Chakra JavaScript engine, which powers Internet Explorer. Currently, Chakra is 30% faster than Chrome V8 on SunSpider, the most widely cited JavaScript benchmark. He started the TypeScript team and contributes to the design and implementation of TypeScript.

3 Environments and Scope (repeated from last class)

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

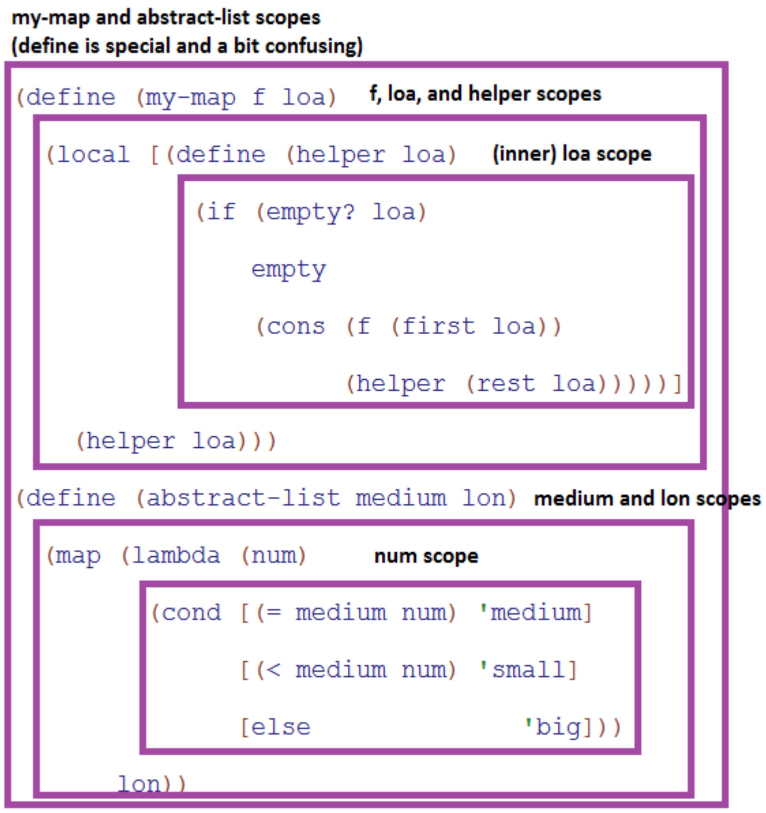
Now that we know where identifier bindings are available, we need a data structure in which to look up identifier references. We'll call that data structure an *environment* because it defines what's "around us" at any given point as our program runs.

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 (or below):

- 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 diagram:


my-map and abstract-list scopes
(define is special and a bit confusing)
f, loa, and helper scopes
(inner) loa scope
medium and lon scopes
num scope

```
(define (my-map f loa)
  (local [(define (helper loa)
            (if (empty? loa)
                empty
                (cons (f (first loa))
                      (helper (rest loa)))))]
    (helper loa)))
(define (abstract-list medium lon)
  (map (lambda (num)
         (cond [(= medium num) 'medium]
               [(< medium num) 'small]
               [else           'big]))
       lon))
```

3.2.1 Why lists for environments?

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.

3.2.2 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
```

4 Environments: Breaking BAD BAD Test Cases

(On an unrelated note, please don't tell me anything from the last season.)

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

file:2013W1-lecture14-environments.rkt has six BAD BAD test cases. They don't currently pass, but we want to make them pass, even though they're bad.

Our goal is to **break** our interpreter in specific ways and learn something about environments from how we break it.

5 What have we learned today?

- From QotD: Dynamic scope really does still get used and really does cause problems (and, perhaps, provide opportunities).
- Scope (repeated from last time)
  - 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 (repeated from last time)
  - 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?!)

- Scope and Environment Implementation and Comprehension
  - Add and manipulate the environment (bindings) available to a program.
  - Give and analyse concrete examples that illustrate the difference between static and dynamic scope.
  - Indicate (some of) the implementation differences between static and dynamic scope.
  - Explain why lazy evaluation with environments can easily lead to the introduction of dynamic scope. (Probably NOT pre-midterm.)

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