Lessons from 6.001  
6.037 - Structure and Interpretation of Computer Programs

Mike Phillips <mpp>  
Massachusetts Institute of Technology

Lecture 8

Contextualizing 6.001

- We said at the start that this wasn’t a class in Scheme
- You’re probably never again going to code in Scheme
- Instead, this is a class in Computation
- How do the concepts from 6.001 apply elsewhere?

Syllabus and key ideas

- Procedural and data abstraction
- Conventional interfaces & programming paradigms
  - Type systems
  - Streams
  - Object-oriented programming
- Metalinguistic abstraction
  - Creating new languages
  - Evaluators

Static scoping is now standard

- Scheme stole static scoping (aka lexical scoping) from ALGOL
- Most languages now are statically scoped, if only by block
- Environment model still describes how bindings work!
Higher-order functions

- Many modern languages support first-class functions:
  - Javascript, Perl, Python, Ruby, MATLAB, Mathematica, C++11, C#, Clojure
- Many even call anonymous functions lambdas

Closures

- Static scoping + first-class functions = closures
- Great for data hiding
  - ... access mediation
  - ... iterators
  - ... continuation passing style for flow control
  - ... laziness or other delayed evaluation

List operations with anonymous functions

- Other languages have filter, map, reduce...
- Map... Reduce?

MapReduce

- Massively parallel architecture for handling Big Data™
- Purely functional code is easy to parallelize — no read/write contention
- Idea based on every call to func in (map func lst) being able to be called in parallel
- ... then also fed into fold-right in parallel
Congratulations, you already know how to write for Hadoop/MapReduce clusters.

What good is writing an evaluator?
- Allows you to move the level of abstraction
- Writing code in Python but need to generate HTML forms?
- Requires programmer have HTML knowledge
- ...or write a Domain-Specific Language (DSL) to generate it for you

External DSLs
- Read and parse a string (syntax)
- Apply arbitrary rules for meaning (semantics)
- We know how to do the latter; there are tools for the former

Internal DSLs
- Can also just write clever function names
- Let your language do the parsing
- Constrains you to the syntax rules of your language
- “For when you want to write code in one language, and get your errors in another!”
Data as code, and vice versa

- Scheme is useful because code and data are just a quote away
- Genetic Programming “evolves” programs by mutating syntax – doable because syntax is simple
- Lisp/Scheme key in early Artificial Intelligence in 1980s
- Useful in deduction languages – which led to PROLOG

Data as code now

- Computers use a language where data is code all of the time
- Assembly language is just bytes
- Data it works on is just bytes

Some random bytes

BF FF FF FF FF 41 80
3C 08 00 75 F9 C3 90

BF FFFFFFFF Store -1 in variable C
41 Add 1 to C
80 3C 08 00 Compare memory at (A + C) to 0
75 F9 If that is not 0, go back 6 bytes
C3 Return
90 Do nothing

When data should not be code

- The most common security vulnerabilities are when computers are convinced that data is actually code
- a.k.a “Buffer overflows”
- Equivalent to making Scheme run an arbitrary function from inside m-eval
- “Jumping out of the system”
Aside: Gödel, Escher, Bach

Evaluators as translators

- Change our evaluator to work in two phases; one parses the expression and returns a Scheme lambda
- The second phase just applies that lambda with a starting environment
- The first phase is a translator from one language to another
- No reason the language we translate to has to be scheme...
- ...how about assembly?

Lowering the abstraction barrier

- Scheme Interpreter (m-eval)
- Scheme Interpreter (Racket)
- Scheme Compiler
- Scheme Interpreter (Racket)
- Assembly
Lowering the abstraction barrier

Transforming from any language to an language

- Now have interpreter in assembly, for Scheme
- How simple a language can we build on?
- Are there functions which can be computed in Java but not Scheme?
- Church-Turing thesis: Turing Machines!

Church-Turing thesis

- If a function can be computed by an algorithm, then it must also be computable by a Turing Machine
- And vice-versa
- Thus Java, Scheme, Python, etc, are all equivalent in the functions they can compute
So if all languages are fundamentally equivalent
... so what do we like about Scheme?
Lexical scoping, procedures as first-class objects, garbage collection, eval and apply, asynchronous event handling...
We have just such a language: Javascript

Brendan Eich was hired by Netscape in 1995 with the promise of “doing Scheme for the browser”
But Java was also being implemented for the browser
So if there was a second language, it should “look like Java”
So syntax closer to Java, but semantics stolen from Scheme
... JavaScript!

Code comparison

Scheme:
(define (make-counter incrementer)
  (let ((counter 0))
    (lambda ()
      (let ((current-val counter))
        (set! counter (incrementer counter))
        current-val))))

Javascript:
function make-counter(incrementer) {
  var counter = 0;
  return function () {
    var current_val = counter;
    counter = incrementer(counter);
    return current_val;
  };
}

And now...

And now for some magic!