Week 10 - Day 1 (Evaluation)

Table of Contents

[CS 403 - 001 Spring 2016](/CS403-001/)

[About](/CS403-001/about/)

# Week 10 - Day 1 (Evaluation)

Mar 22, 2016

# Week 10 - Day 1 Notes

Audio 0:00:38

## Evaluation

### Function calls

function call(pt, env)  
{   
 var args = getArgs(pt);  
 // f will be an identifier  
 var f = getFunction(pt);  
 var closure = eval(f, env);  
 var denv = getEnv(closure);  
 var body = getBody(closure);  
 var params = getParams(closure);  
 // find the value of the arguments in the calling environment  
 // Audio 0:09:47  
 var eargs = evalArgs(args, env);  
 // create a new table  
 var xenv = extend(params, eargs, denv);  
 // Audio 0:11:30  
 // eval the body under the extended environment  
 return eval(body,xenv)  
}

Now that we have the evaluator…

Audio 0:13:30

## Handling Builtins

How do we handle…

println("x is", x);

Println is built in and somewhere in our Java code, we have:

Lexeme evalPrintln(Lexeme pt, Lexeme env) {  
 Lexeme eargs = evalArg(getArgs(pt), env);  
 // then we loop through and basically just print out these things  
 // Audio 0:20:19  
}

Audio 0:19:00 Somehow the above code has to get called by the println code

### Binding Println to Local Environment

var global = extend(null,null,null);  
var prtln = new Lexeme(ID, "println");  
// Audio 0:22:17  
var prtlnVal = new Lexeme(BUILTIN);  
// Audio 0:23:40  
prtlnVal.left = prtln;  
// Audio 0:24:42  
insert(prtln,prtlnVal,global);  
// Now println is in the global scope  
// Audio 0:26:30

(going back to code from function call eval)

// pt = parse tree  
function call(pt, env)  
{   
 var args = getArgs(pt);  
 // f will be an identifier  
 var f = getFunction(pt);  
 var closure = eval(f, env);  
 // Audio 0:27:00  
 // new code!  
 if (closure.type == BUILTIN) {  
 return evalBuiltin(closure, env, args);  
 }  
 // end new code!  
 var denv = getEnv(closure);  
 var body = getBody(closure);  
 var params = getParams(closure);  
 // find the value of the arguments in the calling environment  
 var eargs = evalArgs(args, env);  
 // create a new table  
 var xenv = extend(params, eargs, denv);  
 // eval the body under the extended environment  
 return eval(body,xenv)  
}

Audio 0:29:15

function evalBuiltin(bi, env, args) {  
 var name = bi.left.sval;  
 // Audio 0:30:21  
 if (name == "println") {  
 return evalPrintln(args, env);  
 }  
}

Audio 0:32:04

int ival;  
double nval;  
char \*sval;  
Lexeme \*(\*fval)(lexeme \*, lexeme \*) {  
...  
}

Audio 0:35:00 (I’m lost) ## Hardwiring Builtins Into Grammar Audio 0:36:30

(This is not recommended)

primary: INTEGER  
 | STRING  
 | PRINTLN OPAREN argList CPAREN  
 | ... another builtin

## Handling Operators

Audio 0:39:00

Build a parse tree:

(plus)  
 / \  
 a b

There are also types to worry about. You don’t have to handle all of them, but you have the option of handling int + int, string + string, string + int, etc.

### Plus Eval

Audio 0:42:42

function evalPlus(pt, env) {  
 var a = eval(pt.left, env);  
 var b = eval(pt.right, env);  
 if (a.type == INTEGER && b.type == INTEGER) {  
 return new Lexeme(INTEGER, a.ival + b.ival);  
 }  
 // more types or throw err  
 ...  
}

## Evaluating blocks

Audio 0:44:40

function evalBlock(pt, env) {  
 var spot = getStatements(pt);  
 while (spot != null) {  
 result = eval(spot.left, env);  
 spot = spot.right;  
 }  
 return result;  
}

### Evaluating Return

Audio 0:47:15

function evalReturn(pt, env) {  
 var result = eval(pt, env);  
 var ret = new Lexeme(RETURNED);  
 var left = result;  
 return ret

}

(Coming back to eval block)

function evalBlock(pt, env) {  
 var spot = getStatements(pt);  
 while (spot != null) {  
 result = eval(spot.left, env);  
 // new code!  
 if (result.type == RETURNED) return result;  
 // end new code  
 spot = spot.right;  
 }  
 return result;  
}

### Handling Exceptions

Audio 0:50:00

## Getting Rid of Base Cases

Audio 0:53:00

### Postponing the Evaluation of Arguments

Audio 0:55:00

;@ When we do delayed evaluation, you need the calling environment  
;@ # captures the calling environment  
;@ in scam, if you name a parameter with a $, you delay its evaluation  
 ;@ v-------thunk-------v  
(define (cons-stream # left $right)  
 ;@ Audio 1:00:00  
 (cons left (cons $right #))  
)

### Example Call:

;@ + is only called once  
(cons-stream (+ a b) (+ c d))

Audio 1:01:00

(define (stream-car cell)   
 (car cell)  
)  
  
;@ Audio 1:01:50  
(define (stream-cdr cell)  
 (eval (cadr cell) (cddr cell))  
)

Audio 1:04:00

(What are we talking about?)

## No Base Case

Audio 1:06:09

;@ infinite list of ints  
;@ no base case. We did it guys  
;@ This code is just stored here, it's not being evaluated  
;@ unless we use stream-car or stream-cdr  
(define (ints-from n)  
 (cons n (ints-from (+ n 1))  
 )  
)

Audio 1:08:00

(define two-on (ints-from 2))  
(inspect (stream-car two-on)) ;@ 2  
(inspect (stream-car (stream-cdr two-on))) ;@ 3

## One Last Thing

A stream is these non-evaluated lists

Audio 1:11:59

(define (stream-ref s n)  
 (cond  
 ((= n 0) (stream-car s))  
 (else (stream-ref (stream-cdr s) (- n 1)))  
 )  
)

### Example

(stream-ref two-on 1000) ;@ 1002

## CS 403 - 001 Spring 2016

* CS 403 - 001 Spring 2016
* [jmbeach1@crimson.ua.edu](mailto:jmbeach1@crimson.ua.edu)
* jmbeach

Website for notes and study material for CS 403 (Programming Languages) at The University of Alabama Spring 2016