2018-6-27

Javascript

Project: proj\_part1.zip download

Makefile: runs a bunch of cmds

* make all
* make run
* make test

runs unit tests – get all these to pass

* make solution.zip

creates zip

* make spotless

removes all files

functiondeclrexpr, functionappexpr

language intended for lexical scoping, hook up environment correctly

closureval: function and its environment where it was created

expression.java, environment.java, value.java are where the changes are going to be

expressiontest.java used for unit test

MIDTERM REVIEW

1. AB; hygenic(stomping on var names), syntactic macros(work downstream)
2. ABE
3. AC; lexemes(tokens)
4. –
5. AE; b(caller’s fault), c(can do so on functions), d(opposite)

Louden & lambert’s design criteria

* Efficiency: both machine & coder’s perspective
* Regularity
* Generality
* Orthogonal design
* Uniformity: consistent language design
* Security: do things that aren’t specified(e.g. c arrays)
* Extensibility: ability to grow the language

Racket

* Recursion
* Functional programming language
* List ops
  + Cons
  + Car/cdr
  + List (have an element and want to put it into a list)
  + append

7) largest-elem

(define (largest-elem lst)

(cond [(empty? Lst) (error “empty”)]

[(= (length lst) 1) (car lst)]

[else (let ([x (car lst)]

[max-rest (largest-elem (cdr lst))])

(if (> x max-rest)

X

Max-rest)]))

Syntax

* lexer: text to [tokens]
* parser: [tokens] to AST
* interpreter: uses the AST to sometimes run code
* compiler: produces machine code usually

higher-order functions

* foldl/foldr
* map
* filter

(foldr cons ‘() (‘1 2))

* (cons 1 (foldr con ‘() ‘(2))
* (cons 1 (cons 2 (foldr ‘() ‘()))
* (cons 1 (cons 2 ‘()))
* (cons 1 ‘(2))
* ‘(1 2)

(foldl cons ‘() ‘(1 2))

* (foldl cons (cons 1 ‘()) ‘(2)) ; accumulator(tail-recursive)
* (foldl cons (cons 2 (cons 1’())) ‘())
* (foldl cons ‘(2 1) ‘())
* ‘(2 1)
* (foldl cons ‘(1) ‘(2))
* (foldl cons (cons 2 ‘(1)) ‘())
* (foldl cons ‘(2 1) ‘())
* ‘(2 1)

9) tail-recursive style

(define (mult-all lst)

(ma 1 lst))

(define (ma p lst)

(cond [(empty? lst) p]

[else (ma (\* p (car lst)) (cdr lst))])) ; tail recursive!

* Exports: (provide …)
* Imports: (require “module-name.rkt”)
* Structs
* Hashes: treated as immutable, create a new map with a slight update to change
* Macros[Racket]: syntactic macros, input sequence(pattern), output sequence(template)
* Macros[C]: text-sub macros(C preprocessor), inadvertent variable capture, not hygienic

8) macro replaces syntax, function just does function call

(define-syntax-rule (if-mac c thn els)

(if c thn els))

(define (if-fun c thn els)

(if c thn els))

e.g. if-mac #t (displayln 1) (displayln 0)) ; prints out 1

if-fun #t (displayln 1) (displayln 0)) ; print out 1, then 0, then return void

; because all arguments evaluated before execution

Functional programming langs

* Clearly distinguish inputs from outputs
* No assignment (pure functional)
* No loops (pure functional)
* Results only depends on inputs
* Functions are 1st-class values; anything you do with num/string can be done with functions too
  + Aka referentially transparent
* Tail recursion: can be optimized

Contracts

* BLAME
* Run-time mechanism
* Pre/post conditions: caller/library’s fault
* Contract-out: contracts on the module, more efficient because only runs once
* Define/contract: contracts on the function
  + Benefit: tests internal calls, cost: more checking
* What is a contract?: Function that takes in 1 input and returns t/f
* Contract combinator
  + ->
  + ->I :interplay between various parameters
  + and/c :take a bunch of functions and return t/f based on functions
  + or/c

6)

; better

(provide (contract-out

[balance (-> account? number?)] ; account? account is a struct so free ? function

[deposit (-> account? (and/c number? positive?) account?)])

; long way

(provide (contract-out

[balance (-> account? number?)] ; account? account is a struct so free ? function

[deposit (-> account? positive-num? account?)])

(define (positive-num? n)

(cond [(not number? n)] #f)

[(> n 0)])

Closures

* function
* environment

Scoping

* lexical/static
* dynamic: variable value depends on the path of execution

(define (make-counter)

(let ([count 0])

(lambda ()

(set! count (+ count 1))

count)))

(define my-count (make-counter))

(my-count) ; returns 1

(my-count) ; returns 2 because the environment says count = 1, so (+ count 1)

(define ctr2 (make-counter))

(ctr2) ; returns 1, creates a brand new environment different from my-count

(my-count) ; returns 3

(ctr2) ; returns 2

Operational Semantics

* Big-step
* E v V ; (v is down arrow)
* v1 = v2
* o store ; (o is sigma)
  + mapping from vars to vals
* e, o v V, o’

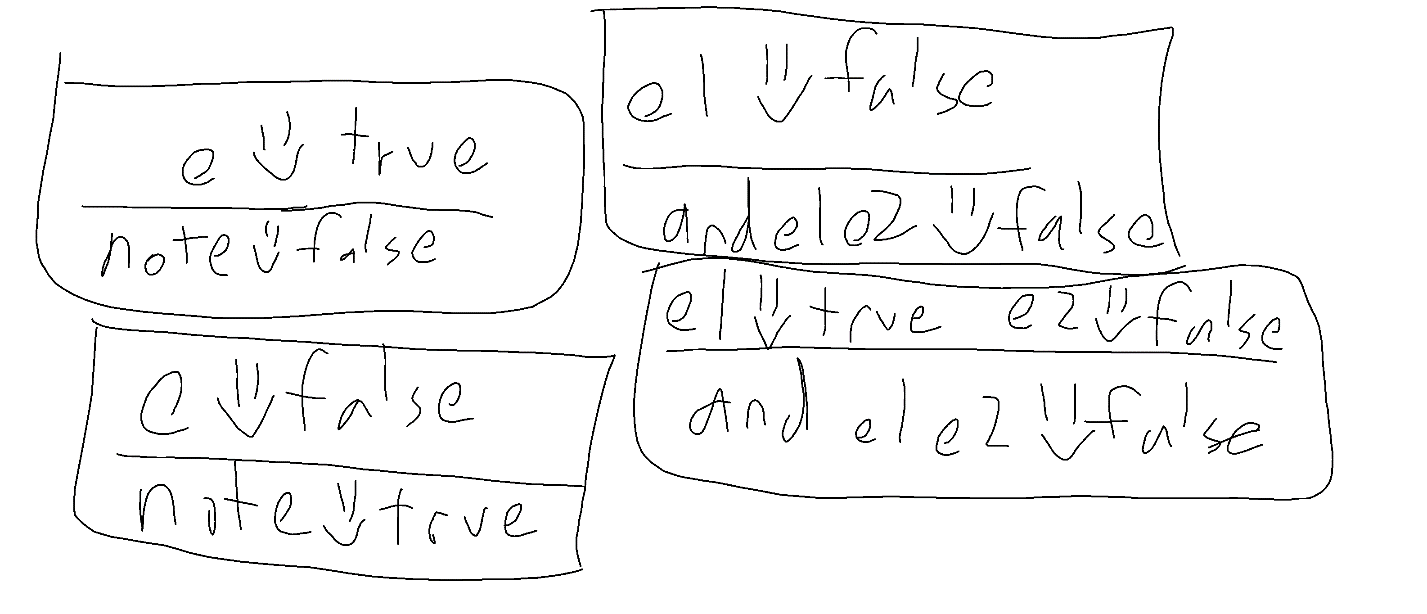
e ::= v

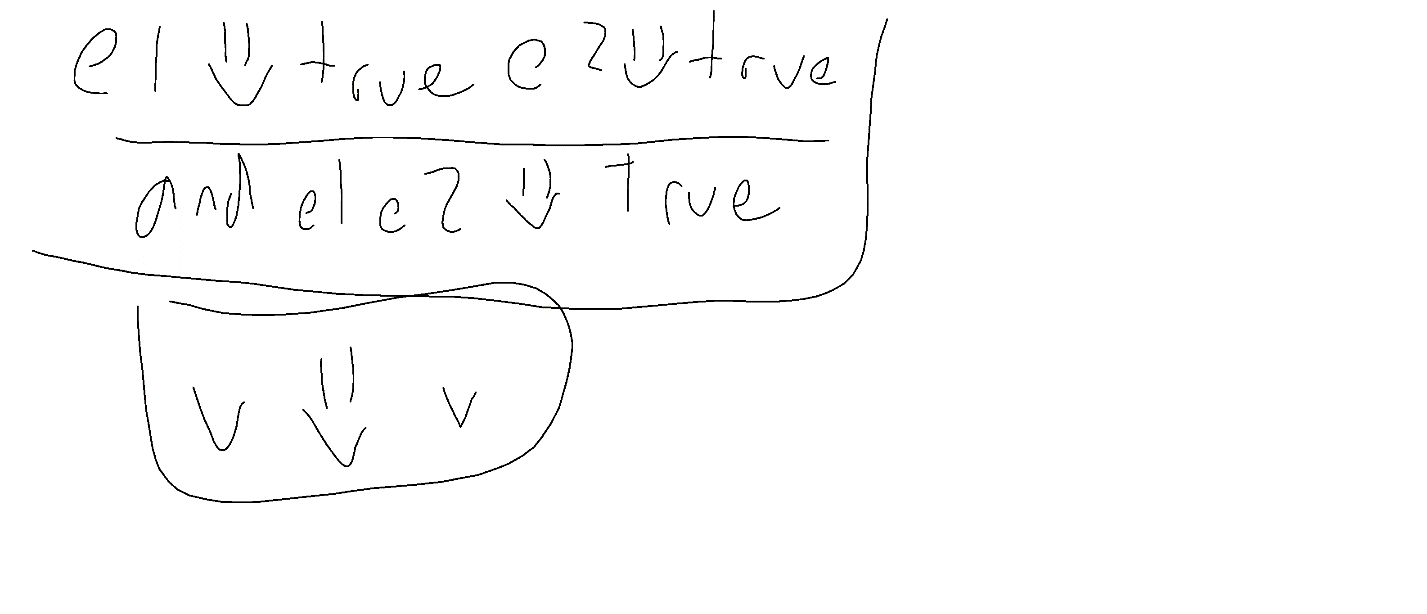
| not e

| and e e

v ::= true

| false





(struct nott (e))

(struct val (v))

(define (evaluate exp)

(match exp [(struct nott (e))

(let ([v (evaluate e)])

(not v))]

(evaluate (nott (val #t)))