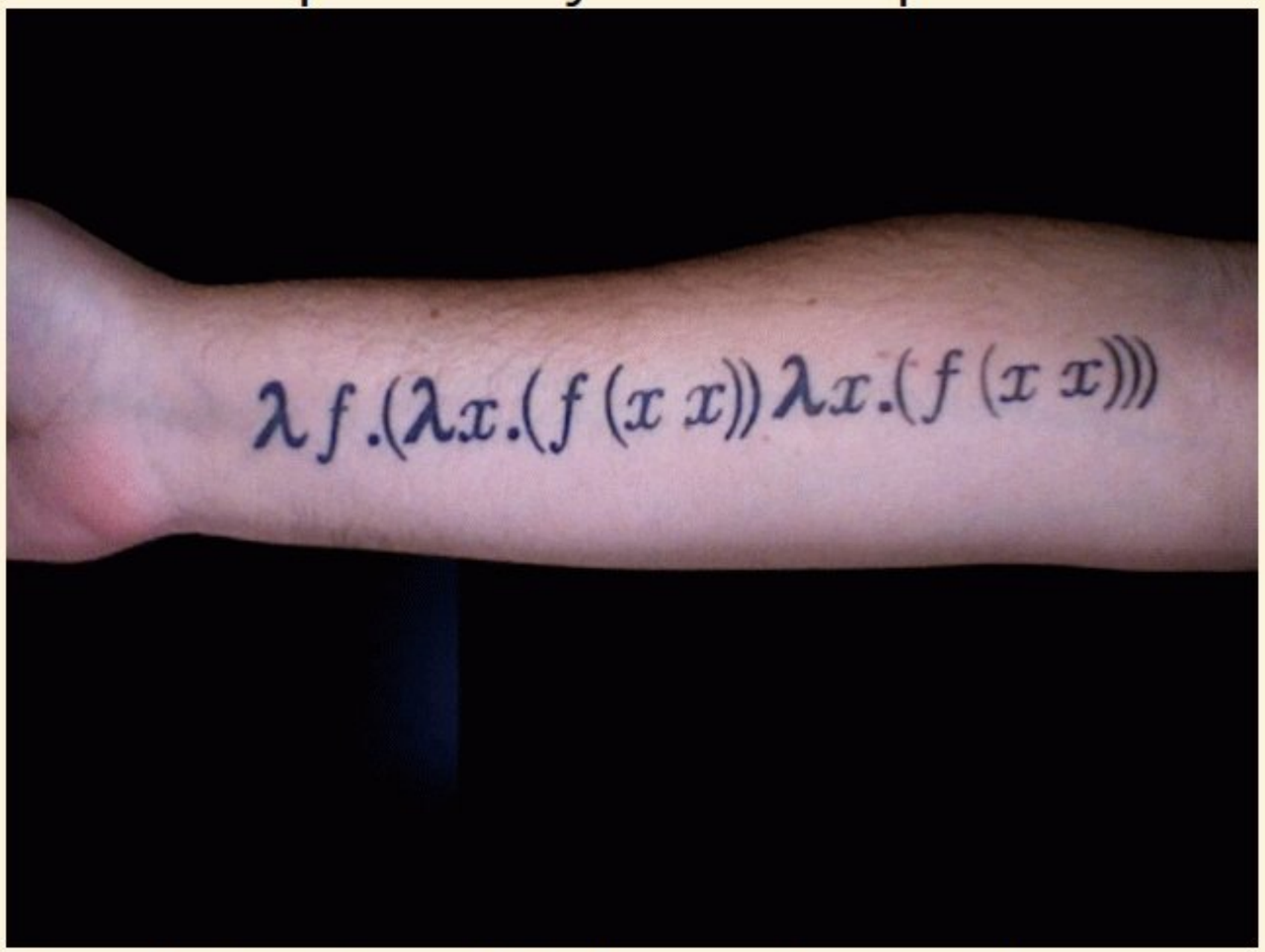
;; _Let's talk about Scheme_

;; * Scheme explicitely builts upon λ calculus



;;

;;

;; * and that's all there is to that

```
;; No, of course not.
;;
;;  * Scheme has special forms
;;        * lambda, define, let, define-syntax, quote ...
;;
;; How would you implement quote?
```

We have to learn them all

* But I don't like to learn so much ...

```
;; _Fexprs_
;;
;; * first-class values
;;     * Arguments are unevaluated ASTs
;;     * Reference to environment on call
;; * Supported until Lisp 1.5 (1958)
;;
;; Later removed in favour of static-analysis
;; optimization possibilities
```

;; Let's do this!
;; http://mainisusuallyafunction.blogspot.de/
;; * Describe a language (Qoppa) built only with Fexprs
;;   * Re-uses few Scheme primitives
;;   * Implements Scheme as library for Qoppa
;;
;; All misunderstandings are mine

```
((vau (x) env x) '(+ 1 2)) # => (+ 1 2)

(define quote
  (vau (x) env
       x))
```

```
(define list (vau xs env
    (if (null? xs)
        (quote ())
        (cons
            (eval env (car xs))
            (eval env (cons list (cdr xs)))))))
```

```
;; _About the Environment_
;; * let's define ourselves an environment
;;   (with frames, and key-value pairs)
;;
(define (bind param val) (cond
    ((and (null? param) (null? val))
        '())
    ((eq? param '_)
        '())
    ((symbol? param)
        (list (list param val)))
    ((and (pair? param) (pair? val))
        (append
            (bind (car param) (car val))
            (bind (cdr param) (cdr val))))
    (else
        (error "can't bind" param val))))
```

```
(define (m-lookup name env)
    (if (null? env)
        (error "could not find" name)
    (let ((binding (assq name (car env))))
        (if binding
            binding
            (m-lookup name (cdr env))))))
```

```
(define (m-eval env exp) (cond
    ((symbol? exp)
        (cadr (m-lookup exp env)))
    ((pair? exp)
        (m-operate env (m-eval env (car exp)) (cdr exp)))
    (else
        exp)))

(define (m-operate env operative operands)
    (operative env operands))
```

```
(define (m-vau static-env vau-operands)
    (let ((params    (car   vau-operands))
          (env-param (cadr  vau-operands))
          (body      (caddr vau-operands)))

        (lambda (dynamic-env operands)
        (m-eval
            (cons
                (bind
                    (cons env-param   params)
                    (cons dynamic-env operands))
                static-env)
            body))))
```

```
;; _Let's run this baby_
;; * We need some kind of bottom "stack frame"
;;     * This includes some necessary + nice-to-have primitives
(define (make-global-frame)
  (define (wrap-primitive fun)
    (lambda (env operands)
      (apply fun (map (lambda (exp) (m-eval env exp)) operands
  (list
   (list 'vau m-vau)
   (list 'eval       (wrap-primitive m-eval))
   (list 'operate    (wrap-primitive m-operate))
   (list 'lookup     (wrap-primitive m-lookup))
   (list 'bool       (wrap-primitive (lambda (b t f) (if b t f
   (list 'eq?        (wrap-primitive eq?))
   (list 'null?      (wrap-primitive null?))
   (list 'symbol?    (wrap-primitive symbol?))
   (list 'pair?      (wrap-primitive pair?))
   (list 'cons       (wrap-primitive cons))
   (list 'car        (wrap-primitive car))
   (list 'cdr        (wrap-primitive cdr))
```

```
(define global-env (list (make-global-frame)))

(define (execute-file filename)
    (let ((stream (open-input-file filename)))
        (define (loop)
            (let ((exp (read stream)))
                (if (eof-object? exp)
                    'done
                    (begin
                        (display exp) (display "\n")
                        (m-eval global-env exp)
                        (loop)))))
        (loop)))

(execute-file "test.qop")
```