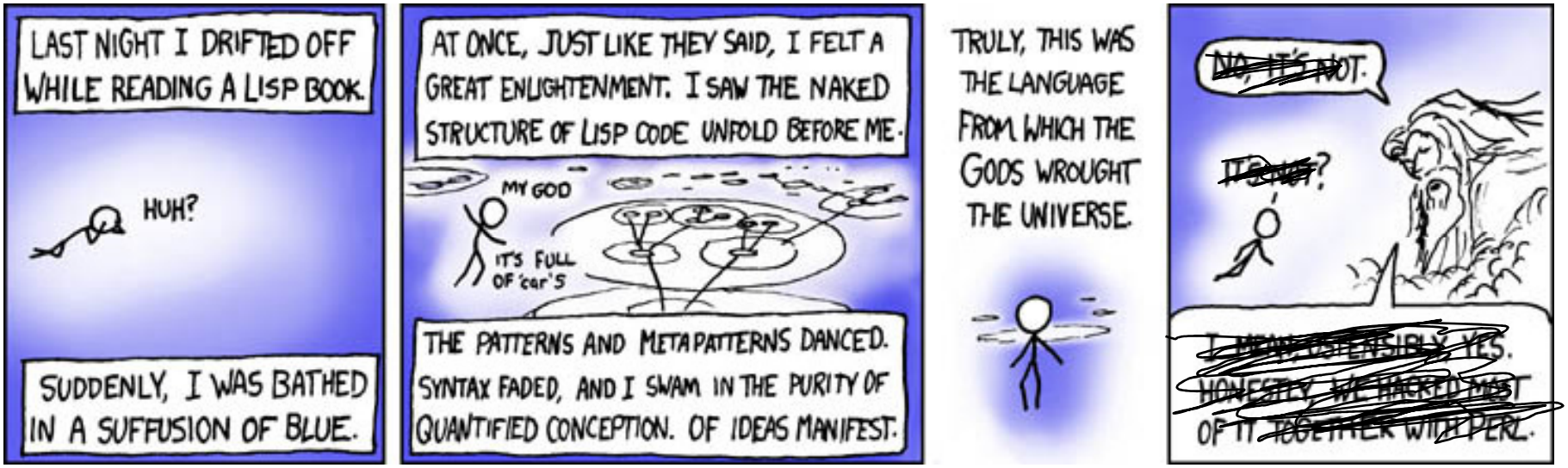
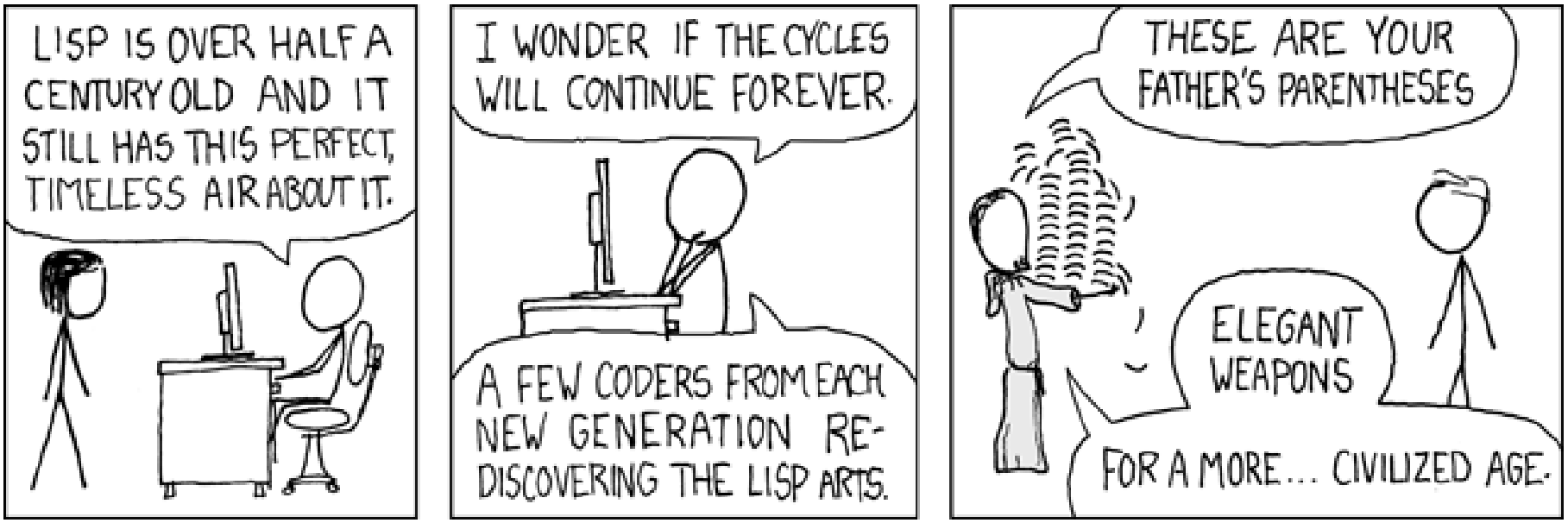
Metacircular Scheme!

(a variant of lisp)

Lisp = Beauty


LAST NIGHT I DRIFTED OFF WHILE READING A LISP BOOK.
HUH?
SUDDENLY, I WAS BATHED IN A SUFFUSION OF BLUE.
AT ONCE, JUST LIKE THEY SAID, I FELT A GREAT ENLIGHTENMENT. I SAW THE NAKED STRUCTURE OF LISP CODE UNFOLD BEFORE ME.
MY GOD
IT'S FULL OF 'car'S
THE PATTERNS AND METAPATTERNS DANCED. SYNTAX FADED, AND I SWAM IN THE PURITY OF QUANTIFIED CONCEPTION. OF IDEAS MANIFEST.
TRULY, THIS WAS THE LANGUAGE FROM WHICH THE GODS WROUGHT THE UNIVERSE.
NO, IT'S NOT.
IT ISN'T?
I MEAN, POSSIBLY. YES.
HONESTLY, WE HACKED MOST OF IT TOGETHER WITH PERL.

Passed on through ages



Basic Expressions

(*function arg1 arg2* …)

To **evaluate** an **expression**, **apply** the *function* to the *arguments*.

(+ 1 2) ?

=> 3

(sqrt 4) ?

=> 2

Nested Expressions

(*function expr1 expr2* …)

The arguments can themselves be *expressions* (which must be evaluated first), so this is a more correct description.

(+ 1 (sqrt 4)) ?

=> 3

(* (+ 2 3) (+ 5 (- 2 2))) ?

=> 25

The function might also be the result of an expression, but you don't need to worry about that now.

Backtrack to Primitives

A primitive is something that evaluates to itself.

Numbers, **Strings** and **Booleans** are primitives, and you'll see more later.

You know you can stop evaluating a nested expression when all you have are primitives.

(**primitive?** 1)

=> true

(primitive? "foo")

=> true

(primitive? (+ 1 2))

=> true

(primitive? (< 1 2))

=> true

List Related Expressions

List is a function that outputs its arguments as a list.

(list 1 2 3)

=> (1 2 3)

(list (+ 0 1) 2 (* 3 1))

=> (1 2 3)

Notice how the output of list looks just like actual code! (except that we don't yet know how to put functions in lists)

(**cons** 0 (list 1 2 3))

=> (0 1 2 3)

Special Forms

Not everything in scheme is a function applied to arguments.
For example: '**if**' only evaluates either the *consequence* or *alternative* (depending on if the *condition* is true).

(*if condition consequence alternative*)

(if (= 1 2) 3 4) ?

=> 4

(if (= x 0) "divide by zero error" (/ 2 x))

Assume x equals 1?

=> 2

Assume x equals 0?

=> "divide by zero error"

Back to Lists

(**first** (list 1 2 3)) ?

    => 1

(**rest** (list 1 2 3)) ?

    => (2 3)

(**null**? (list 1 2 3))

    => false

(null? (list))

    => true

Is list a special form?

No – because all of its arguments are evaluated.

Symbols

A **symbol** is an abstract ‘thing’ that only represents itself.

When you evaluate an expression, you have a dictionary of symbols like ‘+’ which correspond to functions which actually mean something (like adding).

Sometimes symbols have an entry in a dictionary, and sometimes they don’t.

Basic Quoting

Quoting an expression returns that expression, unevaluated. So if the expression is just a function, quoting returns a symbol that looks like that function.

(*quote expression*)

(quote +) ?

=> +

(quote sqrt) ?

=> sqrt

(quote blurppp) ?

=> blurppp

More Quoting

If you quote a number, it simply returns that number.

(quote 2)

=> 2

If you quote a list, it simply returns that list, with all the list elements as symbols.

(quote (a b c)) ?

=> (a b c)

(quote (+ 1 2))

=> (+ 1 2)

Syntactic Sugar

Generally, the philosophy of scheme is to minimize syntax, but occasionally an operation is so common that it gets a shortcut notation (this is known as **syntactic sugar**).

(quote foo) gets abbreviated to ‘foo

(*quote expression*) = *‘expression*

‘(a (quote b) ‘c (+ 1 2))
=> (a ‘b ‘c (+ 1 2))

Defining Functions

(*define* (*function-name arg1 arg2* …)
    expression)

```
(define (square x)
    (* x x))

(square 3)
    => 9
```

Is define a special form?

Yes – it doesn't actually evaluate anything. The function name and arguments are all just names, the expression is only evaluated when the function is called.

Lets Implement Scheme!

We now know enough to start writing our own scheme **interpreter** (also known as an **evaluator**) using scheme.

An interpreter takes in a quoted string (so everything in it is just a valueless symbol), and interprets it as scheme code (uses a ‘dictionary’ to figure out how to evaluate an expression) .

We’ll develop in stages, rewriting what we had before to add more capability to our interpreter.

Start Primitive

```
(define (eval expr)
    (if (primitive? expr)
            expr
            “Not a primitive”))

(eval ‘5)
    => 5
(eval ‘“foo”)
    => “foo”
(eval ‘(1 2))
    => “Not a primitive”
```

Basic Expressions

```
(define (eval expr)
      (if (primitive? expr)
            expr
            (eval-func (first expr) (rest expr))))

(define (eval-func function args)
      (if (equal? function '+)
            (+ (first args) (second args))
            “Function not defined”))

(eval-func ‘(+ 2 1))
      => 3
(eval (- 2 1))
      => 1 ;notice that there is no quote, so it is evaluated by scheme instead of your interpreter
(eval-func ‘(- 2 1))
      => “Function not defined”
```

More Syntactic Sugar

We want more than one function, but nesting 'if's is ugly, so lets use some more of scheme's syntactic sugar: **cond**.

```
(cond (condition1 consequence1)
        (condition2 consequence2)
        …
        (else default-consequence))
```

```
(cond ((< x 0) 0)
        ((> x 1) 2)
        (else 1))
```

Assume x = -1 ?

=> 0

Assume x = .5 ?

=> 1

More Basic Expressions

```
(define (eval-func function args)
    (cond (equal? function '+)  (+ (first args) (second args))
          (equal? function ‘/)  (/ (first args) (second args))
          (equal? function ‘<)  (< (first args) (second args))
          ((equal? function ‘list)  args)
          (else “Function not defined”)))

(eval ‘(+ 3 2))
    => 5
(eval ‘(list 1 2 3))
    => (1 2 3)
(eval ‘(+ 3 (+ 2 1)))
    =>  !!Error – You cannot add numbers and lists!!
```

So we need to add nested expressions.

Recursion

Recursion is the idea of functions calling themselves. It allows one to make loops.

```
(define (mystery n)
      (if (= n 1)
            1                                  {Base Case}
            (* n (mystery (- n 1)))))          {Recursive Case}
```

(mystery 1)
      => 1
(mystery 2)
      => 2
(mystery 3)
      => 6
(mystery 4)
      => 24
What does “mystery” do?
      => factorial

List Recursion

```
(define (mystery lst)
    (if (null? lst)
        lst                                          {Base Case}
        (cons (square (first lst))                   {Recursive Case}
              (mystery (rest lst)))))
```

```
(mystery (list))
    => ()
(mystery (list 1))
    => (1)
(mystery (list 1 2))
    => (1 4)
(mystery (list 1 2 3))
    => (1 4 9)
```

What does “mystery” do?

=> square the elements of a list

Higher Order Functions

```
(define (twice func arg)
   (func (func arg)))

(square 2)
   => 4
(twice square 2)
   =>  16
(twice list 1)
   => ((1))
```

Map

```
(define (map func lst)
    (if (null? lst)
            lst
            (cons (func (first lst))
                    (map func (rest lst)))))

(map square (list 1))
    => 1
(map square (list 1 2 3))
    => (1 4 9)
(map list (list 1 2 3))
    => ((1) (2) (3))
```

Nested Expressions

```
(define (eval expr)
  (if (primitive? expr)
          expr
          (eval-func (first expr) (map eval (rest expr)))))

(eval 3)
    => 3
(eval ‘(+ 3 2))
    => 5
(eval ‘(+ 3 (+ 2 1)))
    => 6
(eval ‘(list (+ 1 (+ 1 1)) (list 1)))
    => (3 (1))
```

Special Forms: Quote

```
(define (eval expr)
  (cond ((primitive? expr)
                    expr)
        ((equal? ‘quote (first expr))
                    (second expr))
        (else
                    (eval-func (first expr) (map eval (rest expr))))))

(eval ‘(quote 1))
    => 1
(eval ‘(quote blurrrp))
    => blurrrp
(eval ‘(quote (+ 1 2)))
    => (+1 2)
```

Special Forms: if

```
(define (eval expr)
  (cond ((primitive? expr)
                              expr)
              ((equal? ‘quote (first expr))
                              (second expr))
              ((equal? ‘if (first expr))
                              (eval-if (second expr) (third expr) (fourth expr)))
              (else
                              (eval-func (first expr) (map eval (rest expr))))))

(define (eval-if condition consequence alternative)
  (if (eval condition)
     (eval consequence)
     (eval alternative)))

(eval-if ‘(< 1 2) (+ 1 1) (/ 1 0))
       => 2
(eval ‘(if (< 2 1) 1 2))
       => 2
```

Define

Adding define is a bit more complicated – you need to keep a dictionary of symbols to values which changes over time (as you define more things). This dictionary is called an **environment**, and is also passed into eval.

Whenever you define something, eval adds the definition to the environment given to it.

(eval expr env)

Apply

The other half of define is Apply

```
(define (eval-func function args env)
        (cond ((primitive-function? function)
                (eval-primitive-func function env)
                ((function-defined? function)
                        (apply func args env))
                (else “Function not defined”))))
```

You would then have to write the functions “primitive-function?”, “function-defined?” and “apply.”

Details

There are a details which I was forced to gloss over due to time constraints, but essentially everything written here is valid code (some trivial helper methods, like ‘primitive?’ and ‘first?’ do need to be defined).

If you are interested in learning more about scheme, see http://mitpress.mit.edu/sicp/full-text/book/book.html and http://icampus.mit.edu/xtutor/content/?6001publichints

To specifically read more about evaluators, see:

http://mitpress.mit.edu/sicp/full-text/sicp/book/node75.html