```scheme
;; Done Twice as a "Dojo" at Villiers Park on Thursday 19th March 2015
;; To groups of about 15 ultra-clever teenagers who were thinking about doing Computer Science at university

;; The first group got as far as higher order functions in an hour.

;; The second group went a bit faster, and we had a bit more time, about an hour and a half,
;; and so we got right to iterative-improve and finding square roots of anything using it.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Environment DrRacket (version 6.1)
;; Language R5RS (Revised Revised Revised Revised Revised Report on the Algorithmic Language Scheme)
;; One person sits at the computer, one person helps them, the rest tell them what to do
;; Every time they achieve something significant, rotate audience->copilot->pilot->audience
;; Notes on back of hand:
(define crib '( 2 3 + (+ 2 3) * define lambda square pythag < #t #f if absolute-value average improve-guess make-improve-guess
error good-enough? good-enough-guess))
;; Notes on back of hand:
(define scheme-crib  '( 2 3 + (+ 2 3) * define lambda (square x) (pythag x y) < #t #f if (absolute-value x)))
(define heron-crib   '( "explain-algorithm (/ 9 3)-> 3"
                        (average a b)
                        (improve-guess x)
                        (make-improve-guess n)
                        (error x)
                        (good-enough? x)
                        (good-enough-guess x)))
(define general-crib '( iterative-improve ))
(define cute-crib    '( (square x)
                        (abs x)
                        (make-improve-guess n)
                        (make-good-enough? n tolerance)
                        (iterative-improve guess improve good-enough?)
                        (make-square-root guess tolerance)
                        engineer-sqrt (make-square-root 1.0 0.00000000001)
                        cautious-engineer-sqrt (make-square-root 1 1/100000000000000000000000000000000000000000000)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Introduction to the Lambda Calculus

;; More precisely, an introduction to the algorithmic language Scheme, which is what you get if you start with
;; the lambda calculus and you trick it out with some extra stuff that often comes in handy, true and false and if
;; and define and also some types of numbers, like integers and fractions, and adding, and multiplying.

;; You can build all that stuff starting from scratch with just lambda, and it's a nice thing to do if you want
;; to understand how it all works, but I reckon you're already ok at that sort of thing.

;; So we'll start from something that can do basic arithmetic, and we'll learn how to find square roots of things.

;; This is an evaluator. You can ask it the values of things.

2
3
+

;; We can apply the procedure to the two numbers

(+ 2 3)


;; Can you tell me the square of 333?

(* 333 333)

;; The brackets mean (work out the value of the things in the brackets, and then do the first thing to the other things)

;; So what do you get if you add the squares of 3 and 4?

(+ (* 3 3) (* 4 4))


;; We have procedures for * and + , but if we ask the evaluator what & means, or what square means
;; it will just say 'I have no clue'.

;; It might be nice if we had a procedure for squaring things
```

```scheme
;; How you make a procedure is with this thing called lambda, which is sort of a rewriting sort of thing.

;; Try (lambda (x) (* x x)), which means 'make me a thing which, when I give the thing x, gives me the value of (* x x) instead'

(lambda (x) (* x x))

;; #<procedure>, it says, which is very like what you get when you type in +, and it says #<procedure:+>.

;; So we hope we've made a procedure like + or *

;; How shall we use it to get the square of 333?

((lambda (x) (* x x)) 333)

;; Now obviously, typing out (lambda (x) (* x x)) every time you mean square is not brilliant,
;; so we want to give our little squaring-thing a name.

(define square (lambda (x) (* x x)))

;; Now how do we find the square of 333?

(square 333) ; 110889

;; So lambda is allowing us to make new things, to turn complicated procedures into simple things

;; and define is allowing us to give things names

;; So now let's make a procedure that takes two things, and squares them both,
;; and adds the squares together, and let's call it pythag

(define pythag
  (lambda (x y)
    (+ (square x) (square y))))

(pythag 3 4)

;; OK, great, now can you figure out how the procedure < works?

( < 3 4)
( < 4 3)
( < 3 4 6)
( < 3 4 2)

;; Notice that these #t and #f things are things that the evaluator knows the value of:
;; They're called true and false.

#f
#t

;; So now the last piece of the puzzle:

;; if takes three things:

(if #t 1 2) ;1
(if #f 1 2) ;2

;; So we've got numbers and *,+,-,/, and we've got #t #f and if, and we've got lambda, and define

;; And so all the stuff we've got above, we can think of it as a reference manual for a little language.

;; We can build the whole world out of this little language.

;; That's what God used to build the universe, and any other universes that might have come to His mind.
;; And we can use it too.

;; Here's the manual

2
*
(* 2 3)
(define forty-four 44)
forty-four
(lambda (x) (* x x))
((lambda (x) (* x x)) 3)
```

```scheme
(if (< 2 3) 2 3)

;; And if we understand these few lines, then we understand the whole thing, and we can fit the little pieces together like this:

(define square (lambda (x) (* x x)))
(square 2)

;; So now I want you to use the bits to make me a function, call it absolute-value, which if you give it a number gives you back
;; the number, if it's positive, and minus the number, if it's negative.

(define absolute-value (lambda (x) (if (> x 0) x (- x))))

(absolute-value 1)
(absolute-value -3)
(absolute-value 0)

;; So I've taught you most of the rules for Scheme, which is a sort of super-advanced lambda calculus, and so if you understand
;; the bits above, then you've got the hang of the lambda calculus plus some more stuff.

;; And it's a bit like chess. The rules of chess are super-simple, you can explain them to babies,
;; like Dr Polgar did to Judit and her sisters.
;; But that doesn't make the babies into good chess players yet. They have to practise.

;; How are we doing for time? We've done the whole of the lambda calculus, plus some extra bits. We should feel pretty smug.

;; (In both cases, this had taken about 35 minutes)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Let's do a little practice exercise. Like a very short game of chess, now I've explained most of the rules.

;; So once upon a time there was this guy, believe it, called 'Hero of Alexandria'.

;; Or sometimes he seems to have been called 'Heron of Alexandria', like Hero was the short version,
;; like he was sometimes called Jack and sometimes called John.

;; Whatever, Hero invented the syringe, and the vending machine, and the steam engine, and the windmill, and the rocket,
;; and the shortest path theory of reflection of light, and did some theatre stuff,
;; and he was like Professor of War at the big library in Alexandria.

;; You get the impression that if the Alexandrian scene had lasted just a little bit longer,
;; the whole industrial revolution would have kicked off right there, and the Romans would have walked on the moon in about AD400.

;; And we'd all be immortal, and live amongst the stars. So you should take the burning of the Library at Alexandria *very* personally.

;; And one of his things was a way of finding the square roots of numbers,
;; which is so good that it was how people found square roots right up until the invention of the computer.

;; So I'm going to explain that method to you, and you're going to explain it to this computer, and then you can get the computer
;; to calculate square roots for you, really fast. And after that you're only a couple of steps away from cracking the
;; Enigma codes and winning the second world war and inventing the internet and creating an artificial intelligence
;; that will kill us all just 'cos it's got better things to do with our atoms. I'm not joking.

;; So careful.... What I've just given you is the first step on the path that leads to becoming a mighty and powerful wizard.
;; And with great power comes great something or other, you'll find it on the internet, so remember that.

;; PAUSE

;; So imagine you want to find the square root of 9. And you're a bit stuck, so you say to your friend, "What's the square root of nine?", and he says it's three.

;; How do you check?

(* 3 3)

;; Bingo. There's another way to check

(/ 9 3)

;; That's what it means to be the square root of something. If you divide the something by the square root, you get the square root back.
```

```
;; But what if your friend had said "err,.. 2 or something?"

(/ 9 2)

;; Notice that the number you put in is too low, but the number you got back is too high.

;; So Heron says, let's take the average.

;; So we need an average function

(define average (lambda (a b) (/ (+ a b) 2)))

(average 2 (/ 9 2)) ; 3 1/4

;; three and a quarter, that's like a much better guess, it's like you'd found a cleverer friend.

;; so try again.

(average 3.25 (/ 9 3.25)) ; 3.009615...

;; and again

(average 3.0096 (/ 9 3.0096)) ; 3.0000153..

(average 3.0000153 (/ 9 3.0000153)) ; 3.000000000039015

;; So you see this little method makes guesses at the square root of nine into much better guesses.

;; We see that this is kind of a repetitive type thing, and if you see one of those, your first thought should be,
;; I wonder if I can get the computer to do that for me?

;; Can you make a function which takes a guess at the square root of nine, and gives back a better guess?

(define improve-guess (lambda (guess) (average guess (/ 9 guess))))

;; I'd better show you how to format these little functions so that they're easier to read

(define improve-guess
  (lambda (guess)
    (average guess (/ 9 guess))))

;; The evaluator doesn't notice the formatting, and it makes it a bit more obvious what's getting replaced by what.

(improve-guess 4) ; 3 1/8
(improve-guess (improve-guess 4)) ; 3 1/400
(improve-guess (improve-guess (improve-guess 4))) ; 3 1/960800

;; We all know what the square root of nine is, let's look at a more interesting number, two.
;; It's a bit of an open question whether 'the square root of two' is a number, or whether it's just a noise
;; that people make with their mouths shortly after you show them a square and tell them about Pythagoras' theorem.

;; Pythagoras used to have people killed for pointing out that you couldn't write down the square root of two.

;; I've got a bit of a confession to make.

;; Someone's already explained to this computer how to find square roots

(sqrt 9)          ; so far so good!
(sqrt 2)          ; 1.4142135623730951   hmmm. let's check.

(square (sqrt 2)) ; 2.0000000000000004

;; So it turns out that this guy's just said, if you can't come up with the square root of two, just lie, and come up with
something
;; that works, close as dammit.

;; Which is like, bad practice, and tends to lead to Skynet-type behaviour in the long run.

;; So let's see what Hero would have said about it.

;; We need a new function that makes guesses better at being square roots of two.
;; It's a bit dirty, but let's just call that improve-guess as well.

;; That's called redefinition, or 'mutation', and it's ok when you're playing around,
```

```
;; but it's a thing you should avoid when writing real programs, because, you know, Skynet issues.

;; Hell, no-one ever got more powerful by refraining from things.

(define improve-guess
  (lambda (guess)
    (average guess (/ 2 guess))))

;; Anyone make a guess?

(improve-guess 1) ; 1 1/2

;; Any good?

(square (improve-guess 1)) ; 2 1/4

;; How wrong?

(- (square (improve-guess 1)) 2) ; 1/4

;; OK, I want you to notice that we've just done the same thing twice

(define improve-guess-9 (lambda (guess) (average guess (/ 9 guess))))
(define improve-guess-2 (lambda (guess) (average guess (/ 2 guess))))

;; Now whenever you see that you've done the same thing twice, and there's this sort of grim inevitability
;; about having to do it a third time someday, you should think:

;; Hey, this looks like exactly the sort of repetitive and easily automated task that computers are good at.

;; And so now I want you to make me (and this is probably the hard bit of the talk...) a function which
;; I give it a number and it gives me back a function which makes guesses at square roots of the number better.

(define make-improve-guess
  (lambda (n)
    (lambda (guess)
      (average guess (/ n guess)))))

;; And now we can use that to make square root improvers for whatever numbers we like

(define i9 (make-improve-guess 9))

(i9 (i9 (i9 (i9 1)))) ; 3 2/21845


(define i2 (make-improve-guess 2))

(i2 (i2 (i2 (i2 1)))) ; 1 195025/470832

;; The first group got this far in about an hour, which was all we had time for, and then we stopped and I waffled for a bit.

;; Now how good are our guesses, exactly?

(- 2 (square (i2 (i2 (i2 (i2 1))))))

;; We could totally make a function out of that:

(define wrongness (lambda (guess) (- 2 (square guess))))

(wrongness (improve-guess 1)) ; -1/4
(wrongness (improve-guess (improve-guess 1))) ; -1/144
(wrongness (improve-guess (improve-guess (improve-guess 1)))) ; -1/166464
(wrongness (improve-guess (improve-guess (improve-guess (improve-guess 1))))) ; -1/221682772224

;; So we're getting closer! When should we stop? Let's say when we're within 0.00000001

(define good-enough? (lambda (guess) (< (absolute-value (wrongness guess)) 0.00000001)))

(good-enough? (improve-guess (improve-guess 1)))  ; #f

(good-enough? (improve-guess (improve-guess (improve-guess (improve-guess 1))))) ; #t

;; Now, we're doing a bit too much typing for my taste.
```

```scheme
;; What we want to do is to say:

;; I'll give you a guess. If it's good enough, just give it back. If it's not good enough, make it better AND TRY AGAIN.

;; This is the hard bit. We need to make a function that calls itself.

;; Go on, have a go

(define good-enough-guess
  (lambda (guess)
    (if (good-enough? guess) guess
        (good-enough-guess (improve-guess guess)))))


(good-enough-guess 1) ; 1 195025/470832

;; YAY VICTORY!


;; The second group got this far in about an 1hr 10 mins, but they all still seemed keen and we didn't have to stop, so:

;; Now this is as much of the talk as I'd written,
;; but actually we've got the time to go a little bit further, if your brains haven't totally exploded, and you might like the next bit,
;; because it makes a nice punchline to the whole thing:

;; There's a pattern here, and it's called iterative-improve

;; And iterative improvement is everywhere in the world, for instance you probably got shown the Newton-Raphson solver at school,
;; which is a thing which can find roots of all sorts of equations very fast, and it works like this, you have an initial guess, and
;; Newton Raphson is a way of making a guess into a better guess, and you need to know when the answer is good enough so you can stop.

;; Or this morning I had a shower, and I got in the shower and I turned the water on to just a random position and it was too hot, so I turned the handle
;; a bit the other way and it was a bit too cold, so I turned it back just a bit and then it was ok so I stopped.

;; And that's the same pattern, and you see this sort of thing all over, it is how you solve big matrices and so on and so forth.

;; And we have just discovered this pattern, which is kind of a fundamental building block when you're writing programs, like a for loop is another basic pattern.

;; So let's see if we can make a function that takes a guess and a way of improving guesses and a way to tell if we're done yet, and gives us back an answer.

(define iterative-improve
  (lambda (guess improve good-enough?)
    (if (good-enough? guess) guess
        (iterative-improve (improve guess) improve good-enough?))))

(iterative-improve 1 (make-improve-guess 2) good-enough?) ; 1 195025/470832



;; This was where we stopped the second session. Here's some waffle:



;; And I think now you can see that we've abstracted a pattern here that will come in handy for the sorts of things that we're trying to do.

;; That's what this talk has really been about, how to build a language which allows you to solve the problems that you're interested in.

;; So I'd like to tidy up the program that we've just written, and put it into the sort of form that I'd have written it in, if I'd been solving this problem
;; and I'd played around for a bit and found what I thought was a nice expression of the ideas that we've been talking about.

(define square (lambda (x) (* x x)))

(define absolute-value (lambda (x) (if (> x 0) x (- x))))
```

```scheme
(define make-improve-guess
  (lambda (n)
    (lambda (guess)
      (average guess (/ n guess))))) ; this bit is Heron's method

(define make-good-enough?
  (lambda (n tolerance)
    (lambda (guess)
      (> tolerance
         (absolute-value (- n (square guess)))))))

(define iterative-improve
  (lambda (guess improve good-enough?)
    (if (good-enough? guess) guess
        (iterative-improve (improve guess) improve good-enough?))))

(define make-square-root
  (lambda (guess tolerance)
    (lambda (n)
      (iterative-improve guess (make-improve-guess n) (make-good-enough? n tolerance)))))


;; We can use these bits to make the sort of square root we usually find provided:
(define engineer-sqrt (make-square-root 1.0 0.00000000000001 ))

(engineer-sqrt 2)

;; And here's what we might use, if we needed really good square roots for some reason:
(define over-cautious-engineer-square-root (make-square-root 1
1/1000000000000000000000000000000000000000000000000000000000000000000))

(over-cautious-engineer-square-root 2)

;; And I hope you can see this this program is actually built of lots of tiny simple pieces,
;; all of which you can understand, and most of which you'll be able to reuse in other contexts.

;; In particular, iterative-improve is a terribly general thing which you can use in lots of ways.

;; And it might have taken us a while to write, although we wrote it as part of a learning-the-language finger-exercise,
;; but we never have to write it again. It works and it will keep working and we've got in the bank now.

;; Here's the take-home message:

;; If you've got a problem, build yourself a language to solve the problem in.

;; To do that, you need to start with a language that allows you to abstract what you do into simple pieces
;; which are easy to understand, so that you can see that they're right and they aren't too snarled up with
;; the little details of the problem you're working on at the moment.

;; And you need a language that allows you to combine the little pieces easily
;; to make new pieces that solve the problem that you're trying to deal with.

;; And there's a sense in which all computer languages are just this lambda calculus.

;; We've done all this in Scheme, which is lambda calculus plus some extra stuff.

;; There's nothing we've done here that can't be done in python, or in ruby, or in perl or in haskell or in lisp.

;; What distinguishes these languages is what extra stuff has already been added to them.

;; But if a language is good enough, and none of the usual features have actually been taken away,
;; which does happen sometimes, then if there's anything missing that you need you can always add it yourself.

;; And then you can use the language that you have to build the language that you need.

;; In a sense, making languages is itself an iterative improvement process.

;; And the big questions are always:

;; How do we make things better? What's good enough? When are we done?
```

```
;; Postscript


;; I'll show you a trick now. We've been using it all along and nobody noticed,
;; but it's the sort of thing that looks like magic, and I don't like magic unless I can cast the spells myself.

(good-enough-guess 1)   ; 1 195025/470832
(good-enough-guess 1.0) ; 1.4142135623746899

;; This is called 'contagion'. There are really two types of numbers.

;; Numbers that look like 432/123 are called 'exact', or 'vulgar fractions'
;; Numbers that look like 1.4142 are called 'inexact', or 'approximate', or 'floating point', or 'decimal fractions'

;; The first type are the sort of numbers that children learn about in school, and that mathematicians use.

;; And the second type are the sort of numbers that engineers use, and they're actually quite a lot more complicated and fuzzy
;; than the exact type. They just sort of work like 'if it's very close, then it's good enough'.

;; The way most computers think about them, they keep about sixteen digits around, and if you want more than that, tough luck.

;; But for some purposes they're better, for instance they're easier to read, and it's a bit of a matter of taste.

;; If you multiply or add an inexact number to an exact number, the answer is always inexact.
;; You can't unapproximate something.

(/ 1 3)   ; 1/3
(/ 1.0 3) ; 0.3333333333333333

;; We all know that 1/3 isn't really 0.33333333333333

;; Mathematicians worry about that sort of thing. Engineers don't. Sometimes aeroplanes crash. Mostly they don't.
```