Are Scheme functions one liners?

Asked 15 years, 10 months ago Modified 1 year, 11 months ago Viewed 1k times

3

I am following *Structure and Interpretation of Computer Programs* and while attempting to solve Ex 1.3 I arrived at the following code as my first attempt:

```
(define (sumsq a b c)(
(define highest (if (> (if (> a b) a b) (if (> a c) a c)) (if (> a b) a b) (if
(> a c) a c)))
(define second_h (if (> (if (> a b) b a) (if (> a c) c a)) (if (> a b) b a) (if
(> a c) c a)))
(+ (* highest highest) (* second_h second_h)))
```

It was not working and I looked up the solution and found them at SICP Wiki

```
;; ex 1.3
;; implemented using only techniques covered to this point

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

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (largest-two-of-three x y z)
  (if (>= x y)
      (sum-of-squares x (if (>= y z) y z))
      (sum-of-squares y (if (>= x z) x z))))
```

The difference was that I was using multiple statements to define variables and then sum the squares, while the correct way is to define each of my lines as functions.

Are functions in Scheme one liners? Or did I miss the whole thing?

function scheme sicp

Share
Improve this question
Follow

edited May 2, 2012 at 5:20
random
9,954 ● 10 ● 69 ● 84

asked Feb 2, 2009 at 12:00
Rajkumar S
2,521 ● 6 ● 23 ● 29

5 Answers

Sorted by: Highest score (default)

6

You should use proper indenting and line breaks to get an overview over your program flow. Your first proposal then reads like this:

```
(define (sumsq a b c)
  ((define highest
     (if (> (if (> a b) a b)
            (if (> a c) a c))
         (if (> a b) a b)
         (if (> a c) a c)))
   (define second-h
     (if (> (if (> a b) b a)
            (if (> a c) c a))
         (if (> a b) b a)
         (if (> a c) c a)))
   (+ (* highest highest)
      (* second-h second-h)))
```

First thing to notice: the parentheses don't match; there is one more opened than closed. Careful inspection shows that one opening parenthesis in the second line is wrong. This is, incidentally, the one that was somehow dangling at the end of your first line. I would hazard a guess that when you tried to evaluate this, nothing happened, as the reader waited for the end of the statement.

Proper indentation is quite important. I think that SICP doesn't explicitly explain it, although the examples are generally done this way. I found a style guide here.

Second observation: You repeat yourself a lot. In all those nested `if` statements, I am not really sure whether you really got the right values out. Look at the solution you found to see how this can be greatly simplified.

You tried to break up the complexity by giving subresults names. Breaking up complexity is good, but it is generally better to name not the results, but the concepts. Think of what you do, then name these activities. Those are functions, and they constitute the language that you finally almost trivially solve your problem in.

Share Improve this answer Follow

answered Feb 2, 2009 at 15:02

Svante

51.4k ● 11 ● 83 ● 124

Yes, scheme did waited for the end of the statement,I tried to balance the () but finally gave up :(– Rajkumar S Feb 3, 2009 at 12:30

Standard indentation really helps with that. There are editors that can automatically indent correctly, and help you with parenthesis matching. I like emacs for that, but there are others. – Svante Feb 3, 2009 at 13:41

What you wrote (minus one extra paren) is:

```
(define (sumsq a b c)
  (define highest
    (if (> (if (> a b) a b)
           (if (> a c) a c))
      (if (> a b) a b)
      (if (> a c) a c)))
  (define second_h
    (if (> (if (> a b) b a)
           (if (> a c) c a))
      (if (> a b) b a)
      (if (> a c) c a)))
  (+ (* highest highest) (* second_h second_h)))
```

Their solution does separate squares and sum-of-squares into separate functions, but I don't think that that's what's important. Not writing `(+ (* a a) (* b b))` **will** stop you from having to name the two values you're calculating, which will let you write the function as one big expression at the end, but there are bigger things to worry about now.

I think the problem you're having is that your (if...) expressions are too big to easily understand. Notice that there are two patterns which show up many times: `(if (> a b) a b)` and `(if (> a b) b a)`. These are the max and min functions, so it's useful to define them as such:

```
(define (min a b) (if (< a b) a b))
(define (max a b) (if (< a b) b a))
```

This way, you can rewrite your solution as:

```
(define (sumsq a b c)
  (define highest
    (if (> (max a b) (max a c))
      (max a b)
      (max a c)))
  (define second_h
    (if (> (min a b) (min a c))
      (min a b)
      (min a c)))
  (+ (* highest highest) (* second_h second_h)))
```

Simplifying it again gives:

```
(define (sumsq a b c)
  (define highest
    (max (max a b) (max a c)))
  (define second_h
```

```
    (max (min a b) (min a c)))
  (+ (* highest highest) (* second_h second_h)))
```

Notice how this writing is much more easy to reason with, `(max (max a b) (max a c))` is obviously the maximum of `a` `b` and `c`, and can actually be rewritten as `(max (max a b) c)`. Looking at `second_h`, though, it's not obvious that it's correct. What will happen when `a` is the smallest of the three values?

The trick they use in their solution is to first compare `x` and `y`. if `x < y`, then you know that `y` is not the smallest of the three, so it is either highest or second highest. The other number you will want to use is the higher of `x` and `z`, since the lower of those two will be the smallest of the three, which you want to ignore. Similar logic applies when `y < x`.

Share Improve this answer Follow

answered Feb 2, 2009 at 15:16

Henk

3,233 ● 2 ● 19 ● 12

4

One of the ideas of Scheme is `bottom-up programming` where you create a function for each and every conceptual operation. That's the recommended approach in many functional programming languages.

With this approach you end up with a lot of small functions implementing one logical operation on the arguments. That way your code ends up being much more modular and clean.

Share Improve this answer Follow

answered Feb 2, 2009 at 12:05

Baishampayan Ghose

20.6k ● 10 ● 58 ● 60

4

Your solution had this form: (define (func param) (define...) (define...))

But define needs this form: (define (func param) body)

The body is the implementation of the function... what it does, what it returns. Your body was just more definitions, never doing anything. So that explains why your solution was not accepted by the Scheme interpreter.

To answer the question "are scheme functions one-liners?" you need to investigate the "begin" form, which looks like this: (begin (+ 1 1) (+ 2 2)) => 4

In the above example, the result of (+ 1 1) is just thrown way, so you can see that begin only really makes sense when the stuff inside of it has side effects.

You should be aware that some parts of Scheme (notably let and lambda) have implicit begins around their body. So this is valid:

```
(let ((x 1))
  (+ 1 1)
  (+ 2 2))
```

even without a begin. This makes the code simpler to write.

Finally, as you continue learning Scheme, always try to find a way to do something with no begins and no side effects. Especially in the first few chapters of most Scheme books, if you are thinking, "I want to set that variable, then I want to do this, then this..." you are probably trapped in your old way of programming and not doing it the Scheme way. There's nothing wrong with side effects at all, but heavy use of them means you are not really programming Scheme the way it works best.

Share Improve this answer Follow

answered Feb 2, 2009 at 13:39

Jeff Allen

1,471 ● 15 ● 20

Scheme does allow the form: (define (func param) (define...) (define...) body ...) The nested defines are converted into a letrec, so it's equivalent to (define (func param) (letrec ((...) (...)) body...). – Henk Feb 2, 2009 at 14:48

1

Exercise 1.3 asks you to define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers. It's easy to define a procedure like that using the built-in Scheme procedures `square`, `max`, and `min`, but we haven't encountered these procedures at that point in the book yet, so I'd define them as well.

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

(define (max x y)
   (if (> x y) x y))

(define (min x y)
   (if (< x y) x y))

(define (sum-of-highest-squares x y z)
   (+ (square (max x y))
      (square (max (min x y) z))))
```

The `sum-of-highest-squares` procedure works by adding the square of the maximum of x and y (the max of those two is eliminated from being the lowest of the three) and

the square of the maximum of the remaining two (the minimum of x and y, which will be whichever value was left over from the first step), and z.

Note: This is from my blog post SICP Exercises 1.1 - 1.5. There are links that will take you to a lot of the other SICP solutions there as well.

Share

Improve this answer

Follow

edited Jan 6, 2023 at 13:59

answered Jan 18, 2011 at 13:55

Bill the Lizard

405k ● 211 ● 572 ● 889