

# sicp-ex-1.1

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

**sicp-solutions | Next exercise (1.2) >>**

---

- 10
- 12
- 8
- 3
- 6
- Value: a
- Value: b
- 19
- #f
- 4
- 16
- 6
- 16

---

Last modified : 2017-11-09 14:48:06  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**

# sicp-ex-1.2

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (1.1) | sicp-solutions | Next exercise (1.3) >>

```
;; ex 1.2

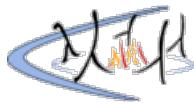
(/ (+ 5
      4
      (- 2 (- 3 (+ 6 (/ 4 5)))))

(* 3
   (- 6 2)
   (- 2 7)))  
;; Result is -0.24666666666666667, or -37/150
```

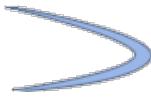
```
;; ex 1.2 bis

(/ (+ 4 5 6 (/ 4 5) (- 2 3))
  (* 3 (- 6 2) (- 2 7)))  
;; The double subtraction 2-(3-(6+4/5)) is simplified to
;; 2-(3-6-4/5) which is then simplified to
;; 2-3+6+4/5 which is better written as
;; 2+6+4/5-3  
;; Now it is -37/150 too (4/5 instead of 4/3)
```

Last modified : 2020-07-14 12:09:08  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-1.3



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (1.2) | sicp-solutions | Next exercise (1.4) >>

Exercise 1.3. Define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers.

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

(define (squareSum x y) (+ (square x) (square y)))

(define (sumOfLargestTwoSquared x y z)
  (cond ((and (>= (+ x y) (+ y z)) (>= (+ x y) (+ x z))) (squareSum x y))
        ((and (>= (+ x z) (+ y z)) (>= (+ x z) (+ x y))) (squareSum x z))
        (else (squareSum y z)))
  ))
```

```
(sumOfLargestTwoSquared 1 2 3)
;Value: 13
(sumOfLargestTwoSquared 1 1 1)
;Value: 2
(sumOfLargestTwoSquared 1 2 2)
;Value: 8
(sumOfLargestTwoSquared 1 1 2)
;Value: 5
(sumOfLargestTwoSquared 1 4 3)
;Value: 25
```

mdsib

Here's a version with simpler logic to detect the smallest of the three values:

```
(define (ssq a b) (+ (* a a) (* b b)))

(define (sumOfLargestTwoSquared x y z)
  (cond ((and (<= x y) (<= x z)) (ssq y z))
        ((and (<= y x) (<= y z)) (ssq x z))
        (else (ssq x y))))
```

master

Another possibility (uses some language constructs that haven't been introduced yet):

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

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

(define (sum-of-squares-of-two-largest x y z)
  (let* ((smallest (min x y z))
         (two-largest (remove smallest (list x y z))))
    (apply sum-of-squares two-largest)))
```

jpp

An even simpler solution

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

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

(define (sum-of-squares-of-two-largest x y z)
  (sum-of-squares (max x y) (max (min x y) z)))
```

jkayser

Another possibility

```
(define (square-two-largest a b c)
```

```
(- (+ (* a a) (* b b) (* c c)) (* (min a b c) (min a b c))))
```

```
gr
(define (square x) (* x x))
(define (sum-of-square a b)
  (+ (square a)
     (square b)))
(define (sum-square-of-two-lager a b c)
  (-
   (+
    (sum-of-square a b)
    c)
   (square (min a b c)))
  )
```

bthomas Here's yet another version.

```
(define (sum-of-squares a b)
  (+ (* a a) (* b b)))

(define (square-of-top a b c)
  (cond ((= a (min a b c)) (sum-of-squares b c))
        ((= b (min a b c)) (sum-of-squares a c))
        ((= c (min a b c)) (sum-of-squares a b))))
```

# sicp-ex-1.4

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

<< Previous exercise (1.3) | sicp-solutions | Next exercise (1.5) >>

---

The `if` statement returns either `a -` or `a +`, which is then applied to the operands.

```
(a + |b|)  
A plus the absolute value of B
```

given

```
(define (a-plus-abs-b a b)  
  ((if (> b 0) + -) a b))
```

```
(a-plus-abs-b 1 -3)  
((if (> -3 0) + -) 1 -3)  
((if #f + -) 1 -3)  
(- 1 -3)  
4  
  
(a-plus-abs-b 1 3)  
((if (> 3 0) + -) 1 3)  
((if #t + -) 1 3)  
(+ 1 3)  
4
```

---

Last modified : 2019-10-06 10:56:10  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-1.5

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

<< Previous exercise (1.4) | sicp-solutions | Next exercise (1.6) >>

---

Using *applicative-order* evaluation, the evaluation of `(test 0 (p))` never terminates, because `(p)` is infinitely expanded to itself:

```
(test 0 (p))  
(test 0 (p))  
(test 0 (p))
```

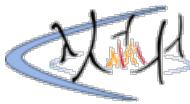
... and so on.

Using *normal-order* evaluation, the expression evaluates, step by step, to 0:

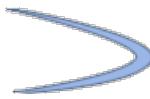
```
(test 0 (p))  
(if (= 0 0) 0 (p))  
(if #t 0 (p))  
0
```

---

Last modified : 2017-11-09 14:50:08  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-1.6



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (1.5) | sicp-solutions | Next exercise (1.7) >>

Exercise 1.6. Alyssa P. Hacker doesn't see why if needs to be provided as a special form. ``Why can't I just define it as an ordinary procedure in terms of cond?'' she asks. Alyssa's friend Eva Lu Ator claims this can indeed be done, and she defines a new version of if:

```
(define (new-if predicate then-clause else-clause)
```

```
  (cond (predicate then-clause)
        (else else-clause)))
```

Eva demonstrates the program for Alyssa:

```
(new-if (= 2 3) 0 5) 5
```

```
(new-if (= 1 1) 0 5) 0
```

Delighted, Alyssa uses new-if to rewrite the square-root program:

```
(define (sqrt-iter guess x)
```

```
  (new-if (good-enough? guess x)
          guess
          (sqrt-iter (improve guess x)
                     x)))
```

What happens when Alyssa attempts to use this to compute square roots? Explain.

The default if statement is a special form which means that even when an interpreter follows applicative substitution, it only evaluates one of its parameters- not both. However, the newly created new-if doesn't have this property and hence, it never stops calling itself due to the third parameter passed to it in sqrt-iter.

To be even clearer: The act of re-defining a special form using generic arguments effectively "De-Special Forms" it. It then becomes subject to applicative-order evaluation, such that any expressions within the consequent or alternate portions are evaluated regardless of the predicate. In Ex 1.6, the iteration procedure is called without return and eventually overflows the stack causing an out of memory error.

```
(define (iff <p> <c> <a>) (if <p> <c> <a>))

(define (tryif a) (if (= a 0) 1 (/ 1 0)))

(define (tryiff a) (iff (= a 0) 1 (/ 1 0)))
```

```
Welcome to DrRacket, version 7.5 [3m].
Language: R5RS; memory limit: 128 MB.
> (tryif 0)
1
> (tryif 1)
. . /: division by zero
> (tryiff 0)
. . /: division by zero
> (tryiff 1)
. . /: division by zero
>
```

(Note: comments below apply to a previous version of this solution, which has been changed to take them into account; please refer to revisions 1–12 of the edit history to view the version on which they were made.)

jsdalton

I believe this solution is incorrect.

new-if does not use normal order evaluation, it uses applicative order evaluation. That is, the interpreter first evaluates the operator and operands and then applies the resulting procedure to the resulting arguments. As with Exercise 1.5, this results in an infinite recursion because the else-clause is always evaluated, thus calling the procedure again ad infinitum.

The if statement is a special form and behaves differently. if first evaluates the predicate, and then

evaluates either the consequent (if the predicate evaluates to `#t`) or the alternative (if the predicate evaluates to `#f`). This is key difference from `new-if` -- only one of the two consequent expressions get evaluated when using `if`, while both of the consequent expressions get evaluated with `new-if`.

wjm

A lenghtier explanation of Applicative Order and Normal Order is here:  
<http://mitpress.mit.edu/sicp/full-text/sicp/book/node85.html>

rdalot

The link is outdated, here is the current working link.  
<https://mitpress.mit.edu/sites/default/files/sicp/full-text/sicp/book/node85.html>

I hope reading that makes the distinction between applicative vs normal order little more clear for others coming here.

dft

But if `if` works the way that you suggest, why does the very first example in wjm's link generate an error?

```
(define (try a b)
  (if (= a 0) 1 b))
```

Evaluating `(try 0 (/ 1 0))` generates an error in Scheme. If `if` only evaluates the consequent or the alternative, it would never get to the division by zero. It seems to me - and this is what the link suggests - that even `if` uses applicative order.

I don't have an alternative explanation - this exercise is stumping me. The applicative vs. normal explanation made sense until I saw the try example above.

dft

Ah, I finally figured it out. You are right. I'm going to keep my question (and this additional response) though because maybe others will have made the same mistake.

The reason the above example generates an error is because `(1 / 0)`, the second parameter to `try`, is evaluated before the `try` is even called. The `if` in the body of `try` is actually irrelevant. An error would be generated even if `try` did not use the value of `b` at all.

As you note, Scheme behaves this way in general due to applicative ordering - parameters are evaluated before the operation is carried out. `if` is an exception where the "parameters" are not evaluated unless needed. So if we say instead:

```
(define (try a)
  (if (= a 0) 1 (/ 1 0)))
```

Calling `(try 0)` does not result in an error, because the else-clause is never evaluated.

andersc

I agree with jsdalton. The reason why `new-if` runs out of memory is applicative order evaluation, so if the plain-old `if` uses applicative order evaluation, it should not work either.

And I guess for a certain interpreter, maybe it should use a consistent way for all processes?

emmp

I believe the above two posters are right and the given answer is wrong.

It's stated clearly in the text that:

"Lisp uses applicative-order evaluation, partly because of the additional efficiency obtained from avoiding multiple evaluations of expressions such as those illustrated with `(+ 5 1)` and `(* 5 2)` above and, more significantly, because normal-order evaluation becomes much more complicated to deal with when we leave the realm of procedures that can be modeled by substitution."

So I don't see a reason why MIT-Scheme (which is supposedly what readers of the book use) would be any different. Plus, as andersc wrote, an interpreter would have to be consistent about the evaluation strategy it uses.

As jsdalton said, `new-if` is a procedure, not a special-form, which means that all sub-expressions are evaluated before `new-if` is applied to the values of the operands. That includes `sqrt-iter` which is extended to `new-if` which again leads to the evaluation of all the sub-expressions including `sqrt-iter` etc. Instead, in `if` only one of the consequent expressions is evaluated each time.

dpchrist

new-if works on my machine.

Here's my code:

```
2013-12-05 21:15:18 dpchrist@desktop ~/sandbox/mit-scheme/sicp2
$ cat ex-1.6.scm | grep -v ';'

(define (average x y) (/ (+ x y) 2))
(define (square x) (* x x))
(define (improve guess x) (average guess (/ x guess)))
(define (good-enough? guess x) (< (abs (- (square guess) x)) 0.001))
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))
(define (sqrt x) (sqrt-iter 1.0 x))
(sqrt 9)
(sqrt (+ 100 37))
(sqrt (+ (sqrt 2) (sqrt 3)))
(square (sqrt 1000))

(define (new-if predicate then-clause else-clause)
  (cond (predicate then-clause)
        (else else-clause)))
(new-if (= 2 3) 0 5)
(new-if (= 1 1) 0 5)
(define (new-sqrt-iter guess x)
  (new-if (good-enough? guess x)
          guess
          (sqrt-iter (improve guess x) x)))
(define (new-sqrt x) (new-sqrt-iter 1.0 x))
(if (= (sqrt 9)
       (new-sqrt 9))
    1 0)
(if (= (sqrt (+ 100 37))
       (new-sqrt (+ 100 37)))
    1 0)
(if (= (sqrt (+ (sqrt 2) (sqrt 3)))
       (new-sqrt (+ (new-sqrt 2) (new-sqrt 3))))
    1 0)
(if (= (square (sqrt 1000))
       (square (new-sqrt 1000)))
    1 0)
```

Here's a sample run on Debian 7.2:

```
2013-12-05 21:17:24 dpchrist@desktop ~/sandbox/mit-scheme/sicp2
$ cat ex-1.6.scm | grep -v ';' | mit-scheme -eval
MIT/GNU Scheme running under GNU/Linux
Type `^C' (control-C) followed by `H' to obtain information about interrupts.

Copyright (C) 2011 Massachusetts Institute of Technology
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Image saved on Saturday October 15, 2011 at 10:11:41 PM
  Release 9.1 || Microcode 15.3 || Runtime 15.7 || SF 4.41 || LIAR/i386 4.118
  Edwin 3.116

1 ]=>
;Value: average

1 ]=>
;Value: square

1 ]=>
;Value: improve

1 ]=>
;Value: good-enough?

1 ]=>
;Value: sqrt-iter

1 ]=>
;Value: sqrt

1 ]=>
;Value: 3.00009155413138

1 ]=>
;Value: 11.704699917758145
```

```

1 ]=>
;Value: 1.7739279023207892

1 ]=>
;Value: 1000.000369924366

1 ]=>
;Value: new-if

1 ]=>
;Value: 5

1 ]=>
;Value: 0

1 ]=>
;Value: new-sqrt-iter

1 ]=>
;Value: new-sqrt

1 ]=>
;Value: 1

1 ]=>
End of input stream reached.
Moriturus te saluto.

```

uninja

The poster above does not define `new-sqrt-iter` as recursive, as it calls the original `sqrt-iter` instead of itself.

srachamim

We can't mimic `if` with `cond` because we can't prevent the interpreter from evaluating specific arguments.

If we use `cond` form instead of `if`, without wrapper it inside `new-if` - it'll still work as expected.

>>>

picard

Read the MIT "Don't Panic" guide to 6.001 on Open Courseware for a short guide on how Edwin works (started with "mit-scheme --edit"). <http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-001-structure-and-interpretation-of-computer-programs-spring-2005/tools/dontpanicnew.pdf>

This exercise is solved by trying out the `new-if` statement and evaluating with M-p in Edwin. You will get an error in the Scheme REPL "Aborting: Maximum recursion depth exceeded" and can look through the debugger to see how `sqrt-iter` loops forever.

trevoriannguyen

I believe the original solution and the comments by previous posters are incorrect. `new-if` is a procedure, and under applicative-order evaluation, **all** its arguments will be evaluated first **before** the procedure application is even started. The third argument to the `new-if` procedure, i.e. the recursive call to `sqrt-iter`, will **always** be evaluated. It is the evaluation of this third argument that causes an infinite loop. In particular, the `else`-clause mentioned by jsdalton is never evaluated. Indeed, the `new-if` procedure body (which contains the `cond` special form) is never even applied to the resulting 3 arguments as the 3rd argument never stops evaluating itself!

student

jsdalton was actually referring to the 3rd argument by its name: `else`-clause. Your statements are thus equivalent.

Shawty Low

I fail to see why `sqrt-iter` is infinitely evaluated in `new-if` but not in the old regular `if`. Haven't we defined a stopping point with `good-enough?` Why should it continue infinitely?

cypherpunkswritecode

Both cond and if are special forms. It's hard to follow, but pay close attention to the wording in SICP.

Page 22: "...there is a special form in Lisp for notating such a case analysis. It is called cond..."

Page 23: "This process continues until a predicate is found whose value is true, in which case the interpreter returns the value of the corresponding consequent expression..."

Take note that it says nothing about evaluating the consequent expression at this point, only returning the value. I believe that the consequent expressions are evaluated first in the case of cond. Now look at the wording for if:

Page 24: To evaluate an if expression, the interpreter starts by evaluating the <predicate> part of the expression. If the <predicate> evaluates to a true value, the interpreter then evaluates the <consequent> and returns its value.

Take note that in the case of if, it explicitly states that the interpreter evaluates the consequent if (and only if) its corresponding predicate is true. That is quite different.

jhenderson

I think, perhaps, the pretty-print is helping hide the elephant in the room here. In sqrt-iter, the call to new-if introduces an infinite recursion. Remember that arguments, if any, are evaluated before a function call. In this case, one of the arguments to new-if invokes sqrt-iter recursively and ad infinitum. The new-if procedure never executes.

poxxa

I think the difference between if and new-if, is new-if's <e> maybe a sequence of expressions.

"A minor difference between if and cond is that the <e> part of each cond clause may be a sequence of expressions." from 1.1.7 of SICP.

I am in curiosity for the difference. Does this may result some bugs?

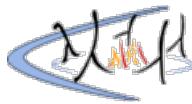
(define (new-if predicate e1 e2)

```
(cond (predicate e1)
      (else e2)))
```

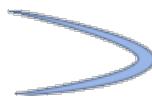
If e1 have a ability to generate "a sequence of expressions".

Then something happened.

(This text is incomplete. It is being worked on incrementally.)



# sicp-ex-1.7



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (1.6) | sicp-solutions | Next exercise (1.8) >>

Maggyero

## QUESTION

The good-enough? test used in computing square roots will not be very effective for finding the square roots of very small numbers. Also, in real computers, arithmetic operations are almost always performed with limited precision. This makes our test inadequate for very large numbers. Explain these statements, with examples showing how the test fails for small and large numbers. An alternative strategy for implementing good-enough? is to watch how guess changes from one iteration to the next and to stop when the change is a very small fraction of the guess. Design a square-root procedure that uses this kind of end test. Does this work better for small and large numbers?

## ANSWER

1. The initial strategy is to stop the improvement of the guess when the absolute error of the guess is less than a constant tolerance. For small radicands, the result is not accurate because the tolerance is not scaled down to the small radicands. For large radicands, the procedure sqrt-iter enters an infinite recursion because the tolerance is not scaled up to the large radicands and floating-point numbers are represented with limited precision so the absolute error at that scale is always greater than the tolerance. The problem observed for large radicands can also be observed for small radicands, providing that the tolerance is chosen so that the absolute error at that scale is always greater than the tolerance.

2. An alternative strategy is to stop the improvement of the guess when the absolute error of the guess is less than a variable tolerance scaled to the radicand, in other words when the relative error of the guess is less than a constant tolerance.

3. Another alternative strategy is to stop the improvement of the guess when the absolute change of the guess is less than a variable tolerance scaled to the guess, in other words when the relative change of the guess is less than a constant tolerance.

Here is a Scheme program implementing the three previous strategies (choose a strategy by using the appropriate good-enough-{strategy-number}? procedure in the sqrt-iter procedure; for a derivation of the minimal tolerance of strategy 2 see <https://math.stackexchange.com/a/3526215/194826>; a similar derivation is used for the minimal tolerance of strategy 3):

```
(import (scheme small))

(define (sqrt x)
  (define (sqrt-iter guess)
    (if (good-enough-3? guess)
        guess
        (sqrt-iter (improve guess))))
  (define (good-enough-1? guess)
    (define tolerance 1.0)
    (< (abs (- (square guess) x)) tolerance))
  (define (good-enough-2? guess)
    (define epsilon (expt 2 -52))
    (define min-float (expt 2 -1022))
    (define tolerance (* 3/2 epsilon))
    (< (abs (- (square guess) x)) (if (= x 0) min-float (* tolerance x))))
  (define (good-enough-3? guess)
    (define epsilon (expt 2 -52))
    (define tolerance (* 9/4 epsilon))
    (or (= guess 0) (< (abs (- (improve guess) guess)) (* tolerance guess))))
  (define (improve guess)
    (/ (+ guess (/ x guess)) 2))
  (define initial-guess 1.0)
  (sqrt-iter initial-guess))

(display (sqrt 0)) (newline)
(display (sqrt 1e-308)) (newline)
(display (sqrt 1e-256)) (newline)
(display (sqrt 1e-128)) (newline)
(display (sqrt 1e-64)) (newline)
(display (sqrt 1e-32)) (newline)
(display (sqrt 1e-16)) (newline)
(display (sqrt 1e-8)) (newline)
(display (sqrt 1e-4)) (newline)
(display (sqrt 1e-2)) (newline)
```

```
(display (sqrt 1e-1)) (newline)
(display (sqrt 1)) (newline)
(display (sqrt 1e1)) (newline)
(display (sqrt 1e2)) (newline)
(display (sqrt 1e4)) (newline)
(display (sqrt 1e8)) (newline)
(display (sqrt 1e16)) (newline)
(display (sqrt 1e32)) (newline)
(display (sqrt 1e64)) (newline)
(display (sqrt 1e128)) (newline)
(display (sqrt 1e256)) (newline)
(display (sqrt 1e307)) (newline)
```

Here is a Python program implementing the three previous strategies (choose a strategy and comment out the two other strategies by prepending an hashtag to their lines; for a derivation of the minimal tolerance of strategy 1 see <https://math.stackexchange.com/a/3526215/194826>; a similar derivation is used for the minimal tolerance of strategy 2):

```
import sys

def sqrt(x):
    def sqrt_iter(guess):
        return guess if good_enough_3(guess) else sqrt_iter(improve(guess))
    def good_enough_1(guess):
        tolerance = 1.0
        try:
            return abs(guess**2 - x) < tolerance
        except OverflowError:
            return False
    def good_enough_2(guess):
        tolerance = 3/2 * sys.float_info.epsilon
        try:
            return abs(guess**2 - x) < (
                sys.float_info.min if x == 0 else tolerance * x
            )
        except OverflowError:
            return False
    def good_enough_3(guess):
        tolerance = 9/4 * sys.float_info.epsilon
        return guess == 0 or abs(improve(guess) - guess) < tolerance * guess
    def improve(guess):
        return (guess + x/guess)/2
    initial_guess = 1.0
    return sqrt_iter(initial_guess)

sys.setrecursionlimit(2000)
print(sqrt(0))
print(sqrt(1e-308))
print(sqrt(1e-256))
print(sqrt(1e-128))
print(sqrt(1e-64))
print(sqrt(1e-32))
print(sqrt(1e-16))
print(sqrt(1e-8))
print(sqrt(1e-4))
print(sqrt(1e-2))
print(sqrt(1e-1))
print(sqrt(1))
print(sqrt(1e1))
print(sqrt(1e2))
print(sqrt(1e4))
print(sqrt(1e8))
print(sqrt(1e16))
print(sqrt(1e32))
print(sqrt(1e64))
print(sqrt(1e128))
print(sqrt(1e256))
print(sqrt(1e307))
```

---

The absolute tolerance of 0.001 is significantly large when computing the square root of a small value. For example, on the system I am using, `(sqrt 0.0001)` yields 0.03230844833048122 instead of the expected 0.01 (an error of over 200%).

On the other hand, for very large values of the radicand, the machine precision is unable to represent small differences between large numbers. The algorithm might never terminate because the square of the best guess will not be within 0.001 of the radicand and trying to improve it will keep on yielding the same guess [i.e. `(improve guess x)` will equal `guess`]. Try `(sqrt 1000000000000)` [that's with 12 zeroes], then try `(sqrt 10000000000000)` [13 zeroes]. On my 64-bit intel machine, the 12 zeroes yields an answer almost immediately whereas the 13 zeroes enters an endless loop. The algorithm gets stuck because `(improve guess x)` keeps on

yielding 4472135.954999579 but (good-enough? guess x) keeps returning #f.

If `good-enough?` uses the alternative strategy (a relative tolerance of 0.001 times the difference between one guess and the next), `sqrt` works better both for small and large numbers.

The best result is obtained by letting the expression keep iterating until the `guess` and the next `guess` are equal (no further improvement is possible at the current precision).

```
;original test
;(define (good-enough? guess x)
;  (< (abs (- (square guess) x)) 0.001))

;iterates until guess and next guess are equal,
;automatically produces answer to limit of system precision
(define (good-enough? guess x)
  (= (improve guess x) guess))
```

```
Welcome to DrRacket, version 7.5 [3m].
Language: R5RS; memory limit: 128 MB.
> (root 9)
3.0
> (root 0.0001)
0.01
> (root 1000000000000.0001)
3162277.6601683795
> (root 1000000000000000000000000)
100000000000.0
> (root 10000000000000000000000000000)
10000000000000.0
> (root 0.00000000000001)
3.162277660168379e-007
> (root 0)
0.0
>
```

```
; ; Modified version to look at difference between iterations
(define (good-enough? guess x)
  (< (abs (- (improve guess x) guess))  
    (* guess .001)))
```

```
; ;Alternate version, which adds an "oldguess" variable to the main function.
(define (sqrt-iter guess oldguess x)
  (if (good-enough? guess oldguess)
      guess
      (sqrt-iter (improve guess x) guess
                 x)))

(define (good-enough? guess oldguess)
  (< (abs (- guess oldguess))
    (* guess 0.001)))

(define (sqrt x)
  (sqrt-iter 1.0 2.0 x))
```

[atov]: The above solutions fail for  $x = 0$ . It hangs and never finishes evaluating. Does anybody know why?

[atoy]: Figured out why the procedure hangs on 0. It hangs because when the guess reaches 0, the delta between guess and oldguess can never be less than (\* guess 0.001) because that evaluates to 0. If you change the '<' operator to '<=' , the procedure will properly evaluate 0.

[random person]: I don't see why (\* guess 0.001) is used. Just '0.001' or whatever tolerance desired seems to work fine. It would be nice if someone explained above if there is a reason why the (\* guess 0.001) is better.

[SchemeNewb]: Just using 0.001 is, in effect, doing the same thing as the original program. It basically says "If the difference between this guess and improved guess is less than 0.0001 in absolute terms (as opposed to percent terms) then stop improving." Problem with this is the same as explained up top. For really tiny numbers, it is easy for the total difference between guess and improve guess to be less than .0001 and for the program to stop without actually doing anything. For large numbers, it might take forever to get to where guess and improved guess are less than .0001. So the book asks us to stop the program if improved guess is less than a certain PERCENT of guess. And THAT is what this alternative does. It checks to see how close guess and improved guess are as a percent. It basically says: "figure out how far guess is from improved guess and then see if that amount is less than .1% of guess. If it is, stop the program"

[robwebbjr]: I don't really know how to explain this, but the first example listed above gives the wrong result after six decimal places, e.g., using the first solution, (`sqrt 0.000005`) returns `0.0022365388240630493` on my intel-64 machine (running Ubuntu), whereas the second solution returns `0.002236068027062195` (as does my calculator). I guess (no pun intended) that it has something to do with calling the `improve` function from the

good-enough? function - but I don't know enough yet to say exactly what's going on there. Here is my solution that gives accurate results beyond six decimal places (just like the second solution from above):

```
;Another take on the good-enough? function

(define (good-enough? guess x)
  (< (/ (abs (- (square guess) x)) guess) (* guess 0.0001)))
```

[tnvu]: One way to "watch how guess changes from one iteration to the next and to stop when the change is a very small fraction of the guess" is to see it as a rate of change using the classic  $(X_1 - X_0) / X_0$ . In this case  $X_1 = (\text{improve } \text{guess } x)$  and  $X_0 = \text{guess}$ . This is equivalent to the first solution (multiply the numerator and denominator by  $\text{guess}$ ) but is more explicit about calculating the rate of change.

```
; A guess is good enough when:
;   abs(improved-guess - original-guess) / original-guess < 0.001

(define (good-enough? guess x)
  (< (abs (/ (- (improve guess x) guess)
              guess))
        0.001))
```

[torinmr]: An alternative approach is to stop iteration when the error (i.e.  $\text{abs}(\text{guess}^2 - x)$ ) is less than a given proportion of  $x$ . This only requires changing one line of the original algorithm:

```
(define (good-enough? guess x)
  (< (abs (- x (square guess)))
      (* 0.0001 x)))
```

GWB

I think the hint is in the question: "in real computers, arithmetic operations are almost always performed with limited precision". Given that it's done with limited precision, at some point, `improve` doesn't actually change the guess any more. This is my solution, and my results:

```
(define (good-enough? guess x)
  (= guess (improve guess x)))

(my-sqrt 1000000000000000000000000)
;Value: 10000000000.

(my-sqrt 0.000000000000000000000000)
;Value: .000000003
```

[JESii]: Unfortunately the answer by GWB only works for "exact" results; if the answer is some form of repeating fractional number, then the equal comparison will never succeed, resulting in an infinite loop.

[White\_Rabbit]: I disagree with JESii. I've never seen an infinite loop with GWB's solution, and I've seen it working for all "non exact" results I've tried. As explained in the intro, when `(improve guess x)` hits the machine's precision it will yield the same guess it got as input and GWB's solution will return TRUE.

[Sreeram]: The approach I have taken is to first narrow down to a close answer and then repeatedly funnelling down to as accurate an answer as allowed by the machine precision

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.1))

(define (small-enuf guess x)
  (=<
    (diff (sqr guess) x)
    (diff (sqr (improve guess x)) x)))

(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      (if (small-enuf guess x)
          guess
          (sqrt-iter (improve guess x) x))
      (sqrt-iter (improve guess x) x)))
```

[Chan]: I wonder whether The good-enough? test will be effective for finding the square roots of large numbers. Before large numbers, I implemented The good-enough? test for small numbers. When i implement `(sqrt 0.0001)`, It returns the same value to first solution. It means I can know The good-enough? test is not effective for finding the square roots of small numbers. But, If you see combinations about larger numbers, you will know that the larger number is, the more precise the value has. So, Isn't the good-enough? test be effective for finding square roots of large numbers?

```
(sqrt 0.0001)
;value: 0.03230844833048122

(sqrt 10000)
;value: 100.00000025490743

(sqrt 1000000)
;value: 1000.0000000000118

(sqrt 10000000)
;value: 3162.277660168379

(sqrt 100000000)
;value: 10000.0
```

[Thomas]: SchemeNewb wrote: "Just using 0.001 is, in effect, doing the same thing as the original program." This is not the case at all — the original programme checks that the \*guess squared\* is within 0.001 of the \*radicand\*, whereas the algorithm described by "random person" checks that the \*new guess\* is within 0.001 of the \*former guess\*. This not only works much better than the original algorithm in all cases (both being more accurate with very small numbers and not hanging the programme with very large numbers) but also works better than

```
(* 0.001 guess)
```

for very large numbers because, 0.001 being by definition smaller than the thousandth of any number larger than 1, the lower tolerance forces the algorithm to continue refining the guess. It is indeed, however, inferior for very small numbers because 0.001 is by definition a larger tolerance than the thousandth of any number smaller than 1. One could cover both bases with the following:

```
(define (good-enough? guess x)
(< (abs (- (improve guess x) guess))
  (cond ((< x 1) (* guess 0.001))
        (else 0.001))))
```

[Owen]: I believe people here are discussing in the wrong direction. The real problem here is the procedure "improve". For the original program (which stops calculating when the difference between  $y^2$  and  $x$  is less than a fixed number 0.001). There are two main risks in it. For a small number  $x$ , 0.001 simply might be too large to be a tolerance threshold. For a large number  $x$ , and this is where people get confused, the real reason for hanging is because (improve guess x) never actually improve the result because of the limitation of bits, the "improved guess" will simply be equal to "old guess" at some point, results in ( $-y^2 x$ ) never changes and hence never reach inside the tolerance range. This situation applied to the small number case as well --- if the threshold is to be set extremely small. The second solution, comparing the difference between new guess and old guess, should never care about a specific precision value (or percentage) at all. Since at some point, the difference between new guess and old guess will guarantee to be 0 because the machine will not be able to represent "the averaging between a guess and (/ x guess)" using fix number of bits. Hence, Thomas's solution can be improved by just setting the threshold to 0.

```
(define (good-enough-thomas? guess x)
(= (- (improve guess x) guess) 0))
```

or, simply reference to GWB's solution, which I believe is the best solution, guaranteeing to stop and at the same time, with the best accuracy.

[Thomas]: Good point, Owen. I should've read the whole discussion before posting — my mistake!

berkentekin	My solution:
	<pre>(define (good-enough-alt? guess x) (&lt; (abs (- 1 (/ (improve guess x) guess))) 0.001))</pre>

Jzuken	Yet another simple solution - check for relative tolerance of guess within 1.0001 and 0.9999 or higher precision:
	<pre>(define (good-enough? guess x) (and (&lt; (abs (/ x (square guess))) 1.0001) (&gt; (abs (/ x (square guess))) 0.9999)))</pre>

KoiCrystal	I use C++ to find how guess changes, I want to find square-root of 1*10^13.
------------	---

when I use "float" type, the algorithm gets stuck on yielding 3162277.50(not  
4472135.954999579)

when I use "double" type("double is more precise than "float",double is 64 bits and float is 32 bits), the  
algorithm gets stuck on yielding 3162277.660168, which is closed to the real  
answer(3162277.660168379) but does not achieve the precision 0.001 (now the abs of guess\*guess-x is  
2.40039)

```
//use "float" type
#include <iostream>
#include <cmath>
using namespace std;
int j = 0;
int good_enough(float guess, float x)
{
    if (abs(guess*guess - x) < 0.001)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
float improve(float guess, float x)
{
    cout.setf(ios::fixed,ios::floatfield);
    cout<<j<<"times: "<<guess<<endl;
    return (guess+x/guess)/2;
}
float sqrt_iter(float guess, float x)
{
    if (j == 100)
    {
        return guess;
    }
    else
    {
        if(good_enough(guess,x) == 1)
        {
            return guess;
        }
        else
        {
            j = j+1;
            sqrt_iter(improve(guess,x),x);
        }
    }
}
int main()
{
    float a = 1000000000000000;
    cout<<sqrt_iter(1.0, a);
}
```

```
//use "double" type
#include <iostream>
#include <cmath>
using namespace std;
int j = 0;
int good_enough(double guess,double x)
{
    if (abs(guess*guess - x) < 0.001)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
double improve(double guess, double x)
{
    cout.setf(ios::fixed,ios::floatfield);
    cout<<j<<"times: "<<guess<<endl;
    return (guess+x/guess)/2;
}
double sqrt_iter(double guess,double x)
{
    if (j == 100)
    {
        return guess;
    }
    else
```

```

    {
        if(good_enough(guess,x) == 1)
        {
            return guess;
        }
        else
        {
            j = j+1;
            sqrt_iter(improve(guess,x),x);
        }
    }
int main()
{
    double a = 10000000000000;
    cout<<sqrt_iter(1.0, a);
}

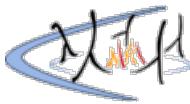
```

then I use relative tolerance of guess within 1.001 and 0.999

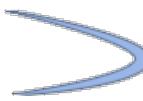
```
(define (new-good-enough? guess x)
  (and (> (/ (square guess) x) 0.999) (< (/ (square guess) x) 1.001)))
```

but the answer is not precise enough(3162433.547242504), until 1.000000001 and 0.999999999, the answer is 3162277.6601683795, nearly to the real answer. so I think the best way is mentioned above by Maggyero: iterating until guess and the next guess are equal

```
(define (good-enough? guess x)
  (= (improve guess x) guess))
```



# sicp-ex-1.8



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (1.7) | sicp-solutions | Next exercise (1.9) >>

Solution using max precision and fix for cube-root of -2 problem. There are still the cube-root of 0 (and cube-root of 100) convergence problem, where the answer approaches 0 (or 100) but never reaches it. (Based on Ex1.7 solution)

```
(define (square guess)
  (* guess guess))

;not used
;(define (average x guess)
;  (/ (+ x guess) 2))

;improve square root
;(define (improve guess x)
;  (average guess (/ x guess)))

;cube root improve formula used as is
(define (improve guess x)
  (/ (+ (/ x (square guess)) (* 2 guess)) 3))

;original test
;(define (good-enough? guess x)
;  (< (abs (- (square guess) x)) 0.001))

;iterates until guess and next guess are equal,
;automatically produces answer to limit of system precision
(define (good-enough? guess x)
  (= (improve guess x) guess))

(define (3rt-iter guess x)
  (if (good-enough? guess x)
      guess
      (3rt-iter (improve guess x) x)))

;<<<expression entry point>>>
;change initial guess to 1.1 to prevent an anomalous result for
;cube root of -2
(define (3root x)
  (3rt-iter 1.1 x))
```

```
Welcome to DrRacket, version 7.5 [3m].
Language: R5RS; memory limit: 128 MB.
> (3root 5)
1.709975946676697
> (3root -2)
-1.2599210498948732
> (3root 27)
3.0
> (3root 0)
4.9406564584125e-324
> (3root 1000000000000000.0001)
46415.88833612779
>
```

The solution presented here is based on the solution for [sicp-ex-1.7](#) and, similarly, uses the alternative strategy for the `good-enough?` predicate.

```
;; ex 1.8. Based on the solution of ex 1.7.

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

(define (cube-root-iter guess prev-guess x)
  (if (good-enough? guess prev-guess)
      guess
      (cube-root-iter (improve guess x) guess x)))

(define (improve guess x)
  (average3 (/ x (square guess)) guess guess))
```

```

(define (average3 x y z)
  (/ (+ x y z) 3))

;; Stop when the difference is less than 1/1000th of the guess
(define (good-enough? guess prev-guess)
  (< (abs (- guess prev-guess)) (abs (* guess 0.001)))))

(define (cube-root x)
  (cube-root-iter 1.0 0.0 x))

;; Testing
(cube-root 1)
(cube-root -8)
(cube-root 27)
(cube-root -1000)
(cube-root 1e-30)
(cube-root 1e60)
;; this fails for -2 due to zero division :(

;; Fix: take absolute cuberoot and return with sign

;;(define (cube-root x)
;;  ((if (< x 0) - +)(cube-root-iter (improve 1.0 (abs x)) 1 (abs x))))

```

```

(define (cube x)
  (* x x x))
(define (improve guess x)
  (/ (+ (/ x (square guess)) (* 2 guess)) 3))
(define (good-enough? guess x)
  (< (abs (- (cube guess) x)) 0.001))
(define (cube-root-iter guess x)
  (if (good-enough? guess x)
    guess
    (cube-root-iter (improve guess x)
      x)))
(define (cube-root x)
  (cube-root-iter 1.0 x))

```

```

(define (cube-root x)
  (cube-root-iter 1.0 x))

(define (cube-root-iter guess x)
  (if (good-enough? guess x)
    guess
    (cube-root-iter (improve guess x)
      x)))

(define (good-enough? guess x)
  (< (relative-error guess (improve guess x)) error-threshold))

(define (relative-error estimate reference)
  (/ (abs (- estimate reference)) reference))

(define (improve guess x)
  (average3 (/ x (square guess)) guess guess))

(define (average3 x y z)
  (/ (+ x y z) 3))

(define error-threshold 0.01)

```

This solution makes use of the fact that (in LISP) procedures are also data.

```

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

(define (good-enough? guess x improve)
  (< (abs (- (improve guess x) guess))
    (abs (* guess 0.001)))))

(define (root-iter guess x improve)
  (if (good-enough? guess x improve)
    guess
    (root-iter (improve guess x) x improve)))

(define (sqrt-improve guess x)
  (/ (+ guess (/ x guess)) 2))

(define (cbrt-improve guess x)
  (/ (+ (/ x (square guess))
        (* 2 guess))
    3))

```

```
(define (sqrt x)
  (root-iter 1.0 x sqrt-improve))

(define (cbrt x)
  (root-iter 1.0 x cbrt-improve))
```

Use the improved good-enough?:

```
(define (cube-roots-iter guess prev-guess input)
  (if (good-enough? guess prev-guess)
      guess
      (cube-roots-iter (improve guess input) guess input)))

(define (good-enough? guess prev-guess input)
  (> 0.001 (/ (abs (- guess prev-guess))
               input))) ;; this should be (abs input) to handle negative inputs. Example: (cube-roots -1) should be -1. Before change, output was 0.33. After fix, output is corrected to -1.00000001794607.

(define (improve guess input)
  (/ (+ (/ input (square guess))
        (* 2 guess))
     3))

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

;;to make sure the first input of guess and prev-guess does not pass the predicate accidentally,
use improve here once:
;;to make sure float number is implemented, use 1.0 instead of 1:
(define (cube-roots x)
  (cube-roots-iter (improve 1.0 x) 1 x))
```

Chan : I just added one procedure. (But I just made this procedure with low precision. I think you can fix this.) Give me a feedback please.

```
(define (cube-root-iter guess x)
  (if (good-enough? guess x)
      guess
      (cube-root-iter (improve guess x) x)))

(define (improve guess x)
  (average (/ x (square guess)) (* 2 guess)))

(define (average x y)
  (/ (+ x y) 3))

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

(define (good-enough? guess x)
  (< (abs (- (cube guess) x)) (* guess 0.001)))

(define (cube x) (* x x x))

(define (cube-root x)
  (if (< x 0)
      (* -1 (cube-root-iter 1.0 (abs x)))
      (cube-root-iter 1.0 x)))

(cube-root 27)
3.0000005410641766

(cube-root -27)
-3.0000005410641766
```

master

I think the following hack may solve the infinite loop problem. I noticed that the reason why calculating the cube root of 100 results in an infinite loop is because once the initial guess has been improved a sufficient number of times it toggles between 4.641588833612779 and 4.641588833612778, which I think are represented identically in binary. I don't think it's possible for more than two numbers to have the same binary representations, so "skipping over" every other number by comparing the guess to a twice improved guess should take care of the issue. Maybe my solution has other issues I'm not aware of or is otherwise flawed but I appear to get the same results as with the original solution.

```
(define (cube x)
  (define (cube-iter guess x)
    (define (improve guess x)
      (/ (+ (/ x (square guess))
            (* 2 guess))
         3))
    (define (good-enough? guess x)
      (= (improve (improve guess x) x) guess)))
```

```
(define (square x) (* x x))
(if (good-enough? guess x)
guess
(cube-iter (improve guess x) x)))
(cube-iter 1.1 x))
```

# sicp-ex-1.9

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

<< Previous exercise (1.8) | sicp-solutions | Next exercise (1.10) >>

---

The process generated by the first procedure is recursive:

```
(+ 4 5)
(inc (+ (dec 4) 5))
(inc (+ 3 5))
(inc (inc (+ (dec 3) 5)))
(inc (inc (inc (+ 2 5))))
(inc (inc (inc (inc (+ (dec 2) 5)))))
(inc (inc (inc (inc (+ 1 5)))))
(inc (inc (inc (inc (+ (dec 1) 5)))))
(inc (inc (inc (inc (+ 0 5)))))
(inc (inc (inc (inc 5))))
(inc (inc (inc 6)))
(inc (inc 7))
(inc 8)
```

9

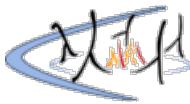
The process generated by the second procedure is iterative:

```
(+ 4 5)
(+ (dec 4) (inc 5))
(+ 3 6)
(+ (dec 3) (inc 6))
(+ 2 7)
(+ (dec 2) (inc 7))
(+ 1 8)
(+ (dec 1) (inc 8))
(+ 0 9)
```

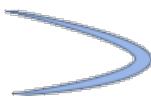
9

---

The easiest way to spot that the first process is recursive (without writing out the substitution) is to note that the "+" procedure calls itself at the end while nested in another expression; the second calls itself, but as the top expression.



# sicp-ex-1.10



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

<< Previous exercise (1.9) | sicp-solutions | Next exercise (1.11) >>

---

(A 1 10)  
(A 0 (A 1 9))  
(A 0 (A 0 (A 1 8)))  
(A 0 (A 0 (A 0 (A 1 7)))))  
(A 0 (A 0 (A 0 (A 0 (A 1 6))))))  
(A 0 (A 0 (A 0 (A 0 (A 0 (A 1 5)))))))  
(A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 1 4)))))))  
(A 0 (A 1 3))))))))  
(A 0 (A 1 2))))))))  
(A 0 (A 1 1))))))))))  
(A 0 (A 2))))))))))  
(A 0 (A 4))))))))))  
(A 0 (A 8))))))))))  
(A 0 (A 16))))))))))  
(A 0 (A 0 (A 0 (A 0 (A 0 (A 32))))))  
(A 0 (A 0 (A 0 (A 0 (A 0 (A 64))))))  
(A 0 (A 0 (A 0 (A 0 (A 128))))  
(A 0 (A 0 (A 256)))  
(A 0 512)  
1024

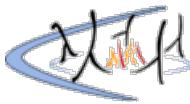
(A 2 4)  
(A 1 (A 2 3))  
(A 1 (A 1 (A 2 2)))  
(A 1 (A 1 (A 1 (A 2 1)))))  
(A 1 (A 1 (A 1 2)))  
(A 1 (A 1 (A 0 (A 1 1)))))  
(A 1 (A 1 (A 0 2)))  
(A 1 (A 1 4))  
(A 1 (A 0 (A 1 3)))  
(A 1 (A 0 (A 0 (A 1 2)))))  
(A 1 (A 0 (A 0 (A 0 (A 1 1))))))  
(A 1 (A 0 (A 0 (A 0 2))))  
(A 1 (A 0 (A 0 4)))  
(A 1 (A 0 8))  
(A 1 16)  
 $2^{16}$   
65536

(A 3 3)  
(A 2 (A 3 2))  
(A 2 (A 2 (A 3 1)))  
(A 2 (A 2 2))  
(A 2 (A 1 (A 2 1)))  
(A 2 (A 1 2))  
(A 2 (A 0 (A 1 1)))  
(A 2 (A 0 2))  
(A 2 4)  
(A 1 (A 2 3))  
(A 1 (A 1 (A 2 2)))  
(A 1 (A 1 (A 1 (A 2 1)))))  
(A 1 (A 1 (A 1 2)))  
(A 1 (A 1 (A 0 (A 1 1)))))  
(A 1 (A 1 (A 0 2)))  
(A 1 (A 1 4))  
(A 1 (A 0 (A 1 3)))  
(A 1 (A 0 (A 0 (A 1 2)))))  
(A 1 (A 0 (A 0 (A 0 (A 1 1))))))  
(A 1 (A 0 (A 0 (A 0 2))))  
(A 1 (A 0 (A 0 4)))  
(A 1 (A 0 8))  
(A 1 16)  
(A 0 (A 1 15))  
(A 0 (A 0 (A 1 14)))  
(A 0 (A 0 (A 0 (A 1 13)))))  
(A 0 (A 0 (A 0 (A 0 (A 1 12)))))  
(A 0 (A 0 (A 0 (A 0 (A 0 (A 1 11)))))))

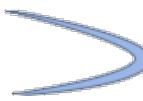
```
(f n) : 2n
(g n) : 0 for n=0, 2^(n) for n>0
(h n) : 0 for n=0, 2 for n=1, 2^(2^(2^(2^(2...(n times)))))) for n>1
```

Alternatively,  $(h_n) : 0 \text{ for } n=0, g^{n-1}(2) \text{ for } n>0$

Last modified : 2023-03-25 16:16:33  
**WiLiKi 0.5-tekili-7** running on **Gauche 0.9**



# sicp-ex-1.11



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (1.10) | sicp-solutions | Next exercise (1.12) >>

The recursive implementation of the given function is straightforward: just translate the definition into Scheme code.

```
;; ex 1.11. Recursive implementation.

(define (f n)
  (cond ((< n 3) n)
        (else (+ (f (- n 1))
                  (* 2 (f (- n 2)))
                  (* 3 (f (- n 3)))))))
```

Note that this solution's use of *cond* and *else* is optional. Since there are only two alternatives, you could use *if* instead of *cond*, without any *else* clause.

## Iterative procedure for Ex 1.11

```
(define (f n)
  (define (f-i a b c count)
    (cond ((< n 3) n)
          ((<= count 0) a)
          (else (f-i (+ a (* 2 b) (* 3 c)) a b (- count 1))))))
  (f-i 2 1 0 (- n 2)))
```

This iterative version will not handle non-integer values while the recursive version will, but as the conditions were given as  $n \geq 3$  and not  $n \geq 3.0$  it is sufficient. Note the  $\leq count 0$  condition. If the condition used = non-integer values would cause an endless loop as *count* would never equal exactly 0. As it is decimal values evaluate to the next whole value. ie 3.2 -> 4

The basic transform is given as:

a <- (a + 2b + 3c)  
b <- a  
c <- b

with a starting condition as defined by the boundary state *f(3)*:

a = 2  
b = 1  
c = 0

and iterating for another  $n-2$  times.

```
Welcome to DrRacket, version 7.6 [3m].
Language: R5RS; memory limit: 128 MB.
> (f -1)
-1
> (f 0)
0
> (f 5)
25
> (f 4.7)
25
> (f 1000)
1200411335581569104197621183222182410228690281055710781687044573790661709343985308756380381850406
6206660426075646316058761566105359337897147801326077556638547442232252494917304286477956022512036
3297367769522100305680356582703510792639565093218070830040971697900925555733636067362640304086340
8122386349183735643342985009827495351241264386090544972951146415009560371824341466875
>
```

The iterative implementation requires a bit of thought. Note that the solution presented here is somewhat wasteful, since it computes  $f(n+1)$  and  $f(n+2)$ .

```
;; ex 1.11. Iterative implementation

(define (f n)
  (define (iter a b c count)
```

```
(if (= count 0)
    a
    (iter b c (+ c (* 2 b) (* 3 a)) (- count 1)))
(iter 0 1 2 n))
```

`;; Testing`

```
(f 0)
(f 1)
(f 2)
(f 3)
(f 4)
(f 5)
(f 6)
```

The above version does not terminate for  $n < 0$ . The following implementation does:

```
(define (f n) (fi n 0 1 2))

(define (fi i a b c)
  (cond ((< i 0) i)
        ((= i 0) a)
        (else (fi (- i 1) b c (+ c (* 2 b) (* 3 a))))))
```

Another implementation, which does not calculate  $f(n+1)$  or  $f(n+2)$ .

```
(define (foo n)
  (define (foo-iter a b c n1)
    ;; a = f(n1 - 1), b = f(n1 - 2), c = f(n1 - 3).
    ;; return a + 2b + 3c
    (if (< n1 3)
        a
        (foo-iter (+ a (* 2 b) (* 3 c)) a b (- n1 1))))
  (if (< n 3)
      n
      (foo-iter 2 1 0 n)))
```

Output

```
> (foo 0)
0
> (foo 1)
1
> (foo 2)
2
> (foo 3)
4
> (foo 4)
11
> (foo 5)
25
> (foo 6)
59
> (foo 7)
142
```

Another iterative version, similar to above, but counting up from 3 to  $n$  (instead of counting down).

```
(define (f n)
  ;; Track previous three values.
  ;; fi-1 is f(i-1)
  ;; fi-2 is f(i-2)
  ;; fi-3 is f(i-3)
  (define (f-iter fi-1 fi-2 fi-3 i)
    ;; Calculate value at current index i.
    (define fi (+ fi-1
                  (* 2 fi-2)
                  (* 3 fi-3)))
    (if (= i n)
        fi
        (f-iter fi fi-1 fi-2 (+ i 1))))
  (if (< n 3)
      n
      (f-iter 2 1 0 3))) ;; start index i=3, count up until reach n.
```

Here is another iterative version that the original poster called "a little bit different".

Another commenter pointed out that it gives wrong answers for  $n < 3$ , but also asked, could someone explain

how this works for larger inputs?

I am not the original author, but after staring at this for a while, I think I can explain it and correct it for  $n < 3$ .

Original version:

```
(define (f n)
  (define (f-iter n a b c)
    ;; this makes f(n) = a f(2) + b f(1) + c f(0) for integer n.
    (if (< n 4)
        ;; N < 4. cause n-1 < 3
        (+ (* a (- n 1))
            (* b (- n 2))
            (* c (- n 3)))
        (f-iter (- n 1) (+ b a) (+ c (* 2 a)) (* 3 a)))
      (f-iter n 1 2 3)))
```

Explanation:

The other iterative versions start from  $f(0)$ ,  $f(1)$ , and  $f(2)$ , and calculate the next  $f(i)$  value based on the previous values.

In contrast, this version tracks just the *coefficients* of  $f(n-1)$ ,  $f(n-2)$ ,  $f(n-3)$ . It starts with coefficients  $(a, b, c) = (1, 2, 3)$ , as given by the definition. It then expands  $f(n)$  in terms of  $f(n-1)$ ,  $f(n-2)$ ,  $f(n-3)$ . And so on, until you get the equivalent value using coefficients for  $f(2)$ ,  $f(1)$ ,  $f(0)$ .

In  $f\text{-iter}$ ,  $n$  is the counter that starts at the given  $n$  and gets decremented until  $n = 3$ . The `if` statement causes execution to stop at  $n = 3$ , and return this expression:

```
(+ (* a (- n 1)) (* b (- n 2)) (* c (- n 3)))
```

Since  $n$  is always 3 at this point (ignore the failure cases of  $n < 3$  for now), that expression is basically:

```
(+ (* a (- 3 1)) (* b (- 3 2)) (* c (- 3 3)))
```

which is:

```
(+ (* a 2) (* b 1) (* c 0))
```

And that satisfies the objective of giving the answer in terms of coefficients of  $f(2)$ ,  $f(1)$ ,  $f(0)$ :

```
(+ (* a f(2)) (* b f(1)) (* c f(0)))
```

So that is why inputs 3 or larger works, while  $n < 3$  fails. For  $n < 3$ , the function should just return  $n$ , rather than calculate coefficients.

Corrected version:

```
(define (f n)
  ;; Given starting coefficients (a, b, c) = (1, 2, 3),
  ;; where f(n) = 1 f(n-1) + 2 f(n-2) + 3 f(n-3),
  ;; f-iter calculates new (a, b, c) such that
  ;; f(n) = a f(2) + b f(1) + c f(0),
  ;; where integer n > 3.
  (define (f-iter n a b c)
    (if (= n 3)
        (+ (* a 2) ;; f(2) = 2
            (* b 1) ;; f(1) = 1 ;; (* b 1) = b, and
            (* c 0)) ;; f(0) = 0 ;; (* c 0) = 0, which can be omitted,
                      ;; but shown here for completeness.
        (f-iter (- n 1) ;; decrement counter
                (+ b a) ;; new-a = a + b
                (+ c (* 2 a)) ;; new-b = 2a + c
                (* 3 a))) ;; new-c = 3a
      ;; main body
      (if (< n 3)
          n
          (f-iter n 1 2 3))))
```

At each step of  $f\text{-iter}$  for  $n$  larger than 3,  $f\text{-iter}$  calls itself with new values for  $a$ ,  $b$ , and  $c$ :

```
new-a = a + b
new-b = 2a + c
new-c = 3a
```

To see where those calculations come from, consider this example of how  $(f 5)$  calculates 25.

```
(f 5)
```

```

(f-iter 5 1 2 3)
n=5, f(5) = 1 f(4) + 2 f(3) + 3 f(2) ;; by definition
= 1 (1 f(3) + 2 f(2) + 3 f(1)) + 2 f(3) + 3 f(2) ;; expand f(4)
= (1 + 2) f(3) + (2 + 3) f(2) + 3 f(1) ;; combine terms
    a + b      2a + c      3a ;; observe pattern
    = 3 f(3)     +      5 f(2) + 3 f(1) ;; new a b c

(f-iter 4 3 5 3)
n=4,      = 3 f(3) + 5 f(2) + 3 f(1) ;; continued
= 3 (1 f(2) + 2 f(1) + 3 f(0)) + 5 f(2) + 3 (f1) ;; expand f(3)
= (3 + 5) f(2) + (6 + 3) f(1) + 9 f(0) ;; combine terms
    a + b      2a + c      3a ;; observe pattern
    = 8 f(2)     +      9 f(1) + 9 f(0) ;; new a b c

(f-iter 3 8 9 9)
n=3,      = 8 f(2) + 9 f(1) + 9 f(0)

;; n=3, so stop looping, and apply a, b, c:
(+ (* a 2) (* b 1) (* c 0))
(+ (* 8 2) (* 9 1) (* 9 0))
(+ 16      9      0)

```

25

Heres another iterative solution, counting up

```

(define (fn-iterate n)
  (define (fn-iter count n f1 f2 f3)
    (if (= count n) f3 (fn-iter (+ count 1) n f2 f3 (+ (* 3 f1) (* 2 f2) f3)))
    (if (<= n 3) n (fn-iter 3 n 1 2 3)))

```

another iterative solution

```

(define (f n)
  (i n 2 1 0))

(define (i n f1 f2 f3)
  (cond ((< n 2) n)
        ((< n 3) f1)
        (else (i (- n 1)
                  (+ f1 (* 2 f2) (* 3 f3)) f1 f2))))

```

Another iterative version

```

(define (f-iterative n)
  (define (sub1 x) (- x 1))
  (define (iter count n-1 n-2 n-3)
    (define (f)
      (+ n-1 (* 2 n-2) (* 3 n-3)))
    (if (= count 0)
        n-1
        (iter (sub1 count) (f) n-1 n-2)))
    (if (< n 3)
        n
        (iter (- n 2) 2 1 0)))

```

And these is how it calculates (f-iterative 7):

```

(f-iterative 7)
(iter (- 7 2) 2 1 0)
(iter (sub1 5) 4 2 1)
(iter (sub1 4) 11 4 2)
(iter (sub1 3) 25 11 4)
(iter (sub1 2) 59 25 11)
(iter (sub1 1) 142 59 25)
142

```

Another iterative version for all integers. Straightforward, but doesn't calculate unneeded values.

```

(define (fi n)
  (define (f-iter a b c count)
    (cond ((< count 0) count)
          ((= count 0) a)
          ((= count 1) b)
          ((= count 2) c)
          (else (f-iter b c (+ c (* 2 b) (* 3 a)) (- count 1)))))
  (f-iter 0 1 2 n))

```

Here are two other iterative versions, designed for readability rather than concision. The first version can handle only integers; the second version can handle integers and numbers with a fractional part. The design decisions promoting readability are:

1. treat the input variable and the counter as separate variables;
2. separate out the logic for calculating the function for a given set of inputs (*theresult* procedure below); and
3. keep the input variables in the same order they appear in the specification.

Neither version does any unnecessary calculations, though they do use an extra variable (by tracking at every iteration both the input, which never changes, and the counter).

### Solution 1 - integers only

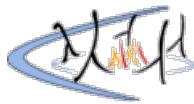
```
(define (f n)
  (define (f-helper n counter a b c)
    (define result
      (+ a (* 2 b) (* 3 c)))
    (cond ((< n 3) n)
          ((>= counter n) result)
          (else (f-helper n (+ 1 counter) result a b))))
  (f-helper n 3 2 1 0))

; initial values, which represent calling function with 3:
; counter = 3
; a = 2 (3-1)
; b = 1 (3-2)
; c = 0 (3-3)
```

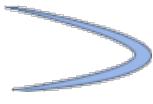
### Solution 2 - integers and numbers with a fractional part

```
(define (f n)
  (define (f-helper n counter a b c)
    (define result
      (+ a (* 2 b) (* 3 c)))
    (cond ((< n 3) n)
          ((>= counter n) result)
          (else (f-helper n (+ 1 counter) result a b))))
  (define fract-n (- n (truncate n)))
  (f-helper n
    (+ 3 fract-n)
    (+ 2 fract-n)
    (+ 1 fract-n)
    (+ 0 fract-n)))
```

To make this approach work for non-integers, the only requirement is to get the fractional part of the input, using the *fract-n* helper function, and update all of the starting values with that fractional part. If the fractional part is zero, solution 1 and solution 2 are identical. Of course, this implementation doesn't account for floating-point inaccuracy, and the result of this called on a floating-point input might differ by a small amount from the result of the recursive version called on the same input.



# sicp-ex-1.12



[\[Top Page\]](#) [\[Recent Changes\]](#) [\[All Pages\]](#) [\[Settings\]](#) [\[Categories\]](#) [\[Wiki Howto\]](#)  
[\[Edit\]](#) [\[Edit History\]](#)  
 Search:

[\*\*<< Previous exercise \(1.11\) | sicp-solutions | Next exercise \(1.13\) >>\*\*](#)

```
;; ex 1.12

(define (pascal r c)
  (if (or (= c 1) (= c r))
      1
      (+ (pascal (- r 1) (- c 1)) (pascal (- r 1) c)))))

;; Testing
(pascal 1 1)
(pascal 2 2)
(pascal 3 2)
(pascal 4 2)
(pascal 5 2)
(pascal 5 3)
```

Computes an entry in the Pascal triangle given the row and column. Rows start from 1, counting from above; columns start from 1 too, counting from left to right.

```
;; ex 1.12

(define (pascal-triangle row col)
  (cond ((> col row) 0)
        ((< col 0) 0)
        ((= col 1) 1)
        ((+ (pascal-triangle (- row 1) (- col 1))
             (pascal-triangle (- row 1) col)))))

;; Testing
(pascal-triangle 1 1)
(pascal-triangle 2 2)
(pascal-triangle 3 2)
(pascal-triangle 4 2)
(pascal-triangle 5 2)
(pascal-triangle 5 3)
```

I find this one easier to grok, it allows only values though:

```
(define (pascal row col)
  (cond ((or (< row col)
             (< col 1)) 0)
        ((or (= col 1)
             (= col row)) 1)
        (else (+ (pascal (- row 1) (- col 1))
                  (pascal (- row 1) col)))))
```

Not having to worry about zero values makes it clearer to me:

```
;; rows / elements numbered starting at 1
;; first / last elements in a row = 1
(define (ptcell r e)
  (cond ((= e 1) 1)
        ((= e r) 1)
        (else (+ (ptcell (- r 1) (- e 1))
                  (ptcell (- r 1) e)))))
```

Off on a tangent having misread the question & skipping ahead a few chapters:

```
;;; calculates nth row of pascal's triangle as a list
(define (pascal n)
  (define (p-row prev)
    (cond ((null? (cdr prev)) (list 1))
          ((= 1 (car prev)) (cons 1 (cons (+ (car prev) (cadr prev)) (p-row (cdr prev))))))
          (else (p-row (cdr prev))))))
```

```

        (else (cons (+ (car prev) (cadr prev)) (p-row (cdr prev))))))
(cond ((< n 1) (display "error: one or more rows"))
      ((= n 1) 1)
      ((= n 2) (list 1 1))
      (else (p-row (pascal (- n 1)))))))

```

If you don't consider out of bounds cases, like negative numbers and other places outside the triangle, you can get a pretty simple solution. Push the triangle so it's left aligned with the 0th column, and you will see that if the column is 0 (in a 0-based world it's the left most of any depth) or if the col equals the depth (the last valid entry in that depth for the triangle), you will get 1. Otherwise, the value of the triangle is the number directly above it added to the number above and to the left.

```

;;Left-aligned triangle, assuming the top most is at (col=0, depth=0)
;;1
;;1 1
;;1 2 1
;;1 3 3 1
;;1 4 6 4 1
;;1 5 10 10 5 1

(define (pascal col depth)
  (cond
    ((= col 0) 1)
    ((= col depth) 1)
    (else (+ (pascal (- col 1) (- depth 1))
              (pascal col (- depth 1)))))))

```

```

; Left-aligned triangale with start at row=0 and col=0
; 1
; 1 5
; 1 4 10
; 1 3 6 10
; 1 2 3 4 5
; 1 1 1 1 1 1
(define (pascal row col)
  (if (or (= row 0) (= col 0))
      1
      (+ (pascal (- row 1) col) (pascal row (- col 1)))))
;; This is wrong. One error is (pascal row (- col 1)) because it means
;; using an element of the same row, which is not what the construction rule
;; says. Another error is not handling out-of-bounds as zero and instead
;; yielding one. This procedure only works for row=0, col=0.

;; This is not wrong! It is an elegant and efficient way from a different perspective.
;; The starting point is at bottom left (0 0), which yields 1 (the root)
;; Every value can be calculated by reading it this way. for example: (pascal 2 2) yields 6
;; If it is a problem that no out-of-bounds are handled,
;; the second row can easily be modified like this:
;; (if (or (< row 1) (< col 1)))

```

Defining pas-n, where:

(pas-n 1) evaluates (pas 1 1)

(pas-n 2) evaluates (pas 2 1)

(pas-n 5) evaluates (pas 3 2)

and so on

```

;; firstly we define (pas x y) the same way as other solutions before
(define (pas x y)
  (if (or (= y 1) (= y x))
      1
      (+ (pas (- x 1) (- y 1))
          (pas (- x 1) y)))))

;;now defining pas-n
(define (pas-n n)
  (define (iter x y counter)
    (cond ((= counter 1) (pas x y))
          ((= x y) (iter (+ x 1) 1 (- counter 1)))
          (else (iter x (+ y 1) (- counter 1)))))
  (iter 1 1 n)))

```

A solution following the [Wikipedia formula](#)

```

(define (pascal n k) ; entry in the nth row and kth column of Pascal's triangle
  (cond ((or (< n 0) ; for any non-negative integer n
             (or (< k 0) (> k n))) ; and any integer k between 0 and n, inclusive
         0) ; treating blank entries as 0
        ((and (= n 0) (= k 0)) 1) ; unique nonzero entry

```

```
(else (+ (pascal (- n 1) (- k 1)) ; adding the number above and to the left
          (pascal (- n 1) k)))) ; with the number above and to the right
```

Here's one with binomial coefficient and tail recursion:

```
(define (pascals-triangle number row)
  (define (factorial n)
    (define (iter n acc)
      (if (< n 2) acc
          (iter (- n 1) (* n acc))))
    (iter n 1))
  (/ (factorial (- row 1)) (* (factorial (- number 1)) (factorial (- (- row 1) (- number 1)))))))
(pascals-triangle 1000 2000)
```

This solution includes some input checks, and errors out if the input is invalid (rather than returning 0 for invalid input, as some solutions above do). It also starts the row and column numbering with zero, as suggested in a [course video](#):

```
(define (pascal row col)
  (cond ((or (= 0 col) (and (= col row) (> col 0))) 1) ; first and last columns
        ((or (< col 0) (< row 0) (> col row)) (raise "bad input")) ; rudimentary input check
        (else
         (+ (pascal (- row 1) (- col 1))
            (pascal (- row 1) col)))))
```

---

Last modified : 2022-11-13 01:15:31  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**

# sicp-ex-1.13

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (1.12) | sicp-solutions | Next exercise (1.14) >>

A simple, beautiful modern solution by Sébastien Gignoux:

<https://codology.net/post/sicp-solution-exercise-1-13/>

Another one by Lucia:

<https://www.evernote.com/shard/s100/sh/6a4b59d5-e99f-417c-9ef3-bcf03a4efecd/7e030d4602a0bef5df0d6dd4c2ad47bf>

This is a solution by Seninha (aka phillbush).

The solution is divided in two proofs, the first one divided in two parts. First, it's proved by induction that  $\text{Fib}(n) = (\varphi^n - \psi^n) / \sqrt{5}$ , this involves two parts: proving for the base case, and proving for the inductive case. Then, that proof is used to prove that  $\text{Fib}(n)$  is the closest integer to  $\varphi^n / \sqrt{5}$ .

The solution uses the  $\vdash$  notation, known as sequent calculus. What is after the  $\vdash$  is what we want to prove. So, for example,  $\vdash A$  means that we want to prove  $A$ ; we will simplify or apply properties to  $A$  during the proof. What comes before the  $\vdash$  are hypotheses that are helpful for our proof. So, for example,  $H_1 \vdash A$  means that we are using the hypothesis  $H_1$  to prove  $A$ . We may use ellipsis to omit previous hypotheses. The symbol  $\top$  means *tautology or truth*.

## Proof 1, part 1 (base case).

We need to prove that  $\text{Fib}(n) = (\varphi^n - \psi^n) / \sqrt{5}$  is valid for  $n=0$  and for  $n=1$ .

$\vdash \text{Fib}(0) = (\varphi^0 - \psi^0) / \sqrt{5} \wedge \text{Fib}(1) = (\varphi^1 - \psi^1) / \sqrt{5}.$

Simplifying both sides of the conjunction.

$\vdash \text{Fib}(0) = 0 \wedge \text{Fib}(1) = 1.$

By definition,  $\text{Fib}(0) = 0$  and  $\text{Fib}(1) = 1$ , so both sides of the conjunction are true.

$\vdash \top \wedge \top.$

Simplifying this conjunction, we prove this part.

$\vdash \top.$

## Proof 1, part 2 (inductive case).

Let  $k$  be a natural number. We are given the two inductive hypothesis  $H_1$  and  $H_2$ , and we need to prove that  $\text{Fib}(n) = (\varphi^n - \psi^n) / \sqrt{5}$  is valid for  $n=k+2$ .

$k : \mathbb{N};$   
 $H_1 : \text{Fib}(k) = (\varphi^k - \psi^k) / \sqrt{5};$   
 $H_2 : \text{Fib}(k+1) = (\varphi^{k+1} - \psi^{k+1}) / \sqrt{5}$   
 $\vdash \text{Fib}(k+2) = (\varphi^{k+2} - \psi^{k+2}) / \sqrt{5}.$

By the definition of  $\text{Fib}$  on the goal, we know that  $\text{Fib}(k+2)$  is equal to  $\text{Fib}(k) + \text{Fib}(k+1)$ . We can rewrite the goal with this fact.

$\dots \vdash \text{Fib}(k) + \text{Fib}(k+1) = (\varphi^{k+2} - \psi^{k+2}) / \sqrt{5}.$

We can rewrite the goal with the hypotheses  $H_1$  and  $H_2$ .

$\dots \vdash (\varphi^k - \psi^k) / \sqrt{5} + (\varphi^{k+1} - \psi^{k+1}) / \sqrt{5} = (\varphi^{k+2} - \psi^{k+2}) / \sqrt{5}.$

We can simplify the left side of the equation on the goal.

$\dots \vdash (\varphi^k (\varphi+1) - \psi^k (\psi+1)) / \sqrt{5} = (\varphi^{k+2} - \psi^{k+2}) / \sqrt{5}.$

As declared in page 38,  $\varphi$  is the golden ratio, the only positive solution to the equation  $x^2 = x + 1$ . The  $\psi$  constant also share that property, being the only negative solution to that equation. We can apply this equation to  $\varphi$  and to  $\psi$ :

$$\dots \vdash (\varphi^k \cdot \varphi^2 - \psi^k \cdot \psi^2) / \sqrt{5} = (\varphi^{k+2} - \psi^{k+2}) / \sqrt{5}.$$

We can simplify the left side of the equation on the goal.

$$\dots \vdash (\varphi^{k+2} - \psi^{k+2}) / \sqrt{5} = (\varphi^{k+2} - \psi^{k+2}) / \sqrt{5}.$$

Both sides of the equation on the goal are equal. We achieved truth.

$$\dots \vdash \top.$$

## Proof 2.

Now that we proved that  $\text{Fib}(n) = (\varphi^n - \psi^n) / \sqrt{5}$ , we can use this fact as hypothesis  $H_1$  to prove that  $\text{Fib}(n)$  is the closest integer to  $\varphi^n / \sqrt{5}$ . Formally, we want to prove that the absolute value of  $\text{Fib}(n)$  minus  $\varphi^n / \sqrt{5}$  is less than  $1/2$ , for all  $n$  natural.

$$\begin{aligned} n &: \mathbb{N}; \\ H_1 &: \text{Fib}(n) = (\varphi^n - \psi^n) / \sqrt{5} \\ \vdash & | \text{Fib}(n) - \varphi^n / \sqrt{5} | < 1/2. \end{aligned}$$

We can rewrite  $\text{Fib}(n)$  on the goal with the hypothesis  $H_1$ .

$$\dots \vdash | (\varphi^n - \psi^n) / \sqrt{5} - \varphi^n / \sqrt{5} | < 1/2.$$

We can simplify the left side of the inequality on the goal.

$$\dots \vdash | \psi^n | / \sqrt{5} < 1/2.$$

We can then apply the definition of  $\psi$ .

$$\dots \vdash | ((1-\sqrt{5})/2)^n | / \sqrt{5} < 1/2.$$

We can simplify the left side of the inequality on the goal.

$$\dots \vdash ((\sqrt{5}-1)/2)^n / \sqrt{5} < 1/2.$$

We can add another hypothesis ( $H_2$ ) for the fact that  $\sqrt{5} < 3$ .

$$\dots; H_2: \sqrt{5} < 3 \vdash ((\sqrt{5}-1)/2)^n / \sqrt{5} < 1/2.$$

We can subtract both sides of the inequality in the hypothesis  $H_2$  by one, then divide both sides by two, raise both sides to the  $n$ -th power, and multiply both sides by  $1/\sqrt{5}$ .

$$\dots; H_2: 1/\sqrt{5} \times ((\sqrt{5}-1)/2)^n < 1/\sqrt{5} \vdash ((\sqrt{5}-1)/2)^n / \sqrt{5} < 1/2.$$

It's a fact that  $1/\sqrt{5} < 1/2$ , we can apply this fact to  $H_2$ .

$$\dots; H_2: ((\sqrt{5}-1)/2)^n / \sqrt{5} < 1/\sqrt{5} < 1/2 \vdash ((\sqrt{5}-1)/2)^n / \sqrt{5} < 1/2.$$

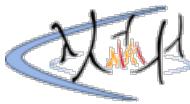
We can fold the double inequality in  $H_2$  by removing the middle part.

$$\dots; H_2: ((\sqrt{5}-1)/2)^n / \sqrt{5} < 1/2 \vdash ((\sqrt{5}-1)/2)^n / \sqrt{5} < 1/2.$$

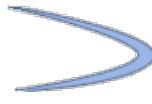
The hypothesis  $H_2$  is exactly what we want to prove, we achieved the truth by redundancy.

$$\dots \vdash \top.$$

QED.



# sicp-ex-1.14



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
 [Edit] [Edit History]  
 Search:

<< Previous exercise (1.13) | sicp-solutions | Next exercise (1.15) >>

Maggyero

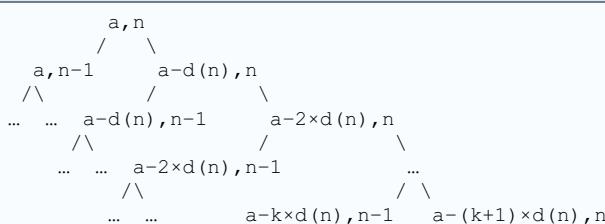
## QUESTION

Draw the tree illustrating the process generated by the count-change procedure of section 1.2.2 in making change for 11 cents. What are the orders of growth of the space and number of steps used by this process as the amount to be changed increases?

## ANSWER

```
(count-change 11)
(cc 11 5)
(+ (cc 11 4) (cc -39 5))
(+ (+ (cc 11 3) (cc -14 4)) 0)
(+ (+ (+ (cc 11 2) (cc 1 3)) 0) 0)
(+ (+ (+ (+ (cc 11 1) (cc 6 2)) (+ (cc 1 2) (cc -9 3))) 0) 0)
(+ (+ (+ (+ (+ (cc 11 0) (cc 10 1)) (+ (cc 6 1) (cc 1 2))) (+ (+ (cc 1 1) (cc -4 2)) 0)) 0) 0)
(+ (+ (+ (+ (+ 0 (+ (cc 10 0) (cc 9 1))) (+ (+ (cc 6 0) (cc 5 1)) (+ (cc 1 1) (cc -4 2)))) (+ (+ (+ (cc 1 0) (cc 0 1)) 0) 0) 0)
(+ (+ (+ (+ 0 (+ 0 (+ (cc 9 0) (cc 8 1)))) (+ (+ 0 (+ (cc 5 0) (cc 4 1))) (+ (+ (cc 1 0) (cc 0 1)) 0))) (+ (+ (+ 0 1) 0) 0) 0)
(+ (+ (+ (+ 0 (+ 0 (+ 0 (+ (cc 8 0) (cc 7 1)))) (+ (+ 0 (+ 0 (+ (cc 4 0) (cc 3 1)))) (+ (+ 0 1) 0)) 1) 0) 0)
(+ (+ (+ (+ 0 (+ 0 (+ 0 (+ 0 (+ (cc 7 0) (cc 6 1)))))) (+ (+ 0 (+ 0 (+ 0 (+ (cc 3 0) (cc 2 1)))) 1)) 1) 0) 0)
(+ (+ (+ (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ (cc 6 0) (cc 5 1)))))) (+ (+ 0 (+ 0 (+ 0 (+ 0 (+ (cc 2 0) (cc 1 1)))) 1)) 1) 0) 0)
(+ (+ (+ (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ (cc 5 0) (cc 4 1)))))) (+ (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ (cc 1 0) (cc 0 1)))) 1)) 1) 0) 0)
(+ (+ (+ (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ (cc 4 0) (cc 3 1)))))) (+ (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 1)))) 1)) 1) 0) 0)
(+ (+ (+ (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ (cc 3 0) (cc 2 1)))))) 2)) 1) 0) 0)
(+ (+ (+ (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ (cc 2 0) (cc 1 1))))))))))) 2) 1) 0) 0)
(+ (+ (+ (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ (cc 1 0) (cc 0 1))))))))))) 2) 1) 0) 0)
(+ (+ (+ (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ 0 (+ (cc 1 0) (cc 0 1))))))))))) 2) 1) 0) 0)
```

Tree of the process for changing an amount  $a$  using  $n$  kinds of coins:



where  $a - k \times d(n) > 0$  and  $a - (k + 1) \times d(n) \leq 0$ , that is  $k = \lceil a/d(n) \rceil - 1$ .

The space required by the process is the height of the tree and grows as  $\Theta(a)$  with  $a$  and  $n$ :

$$R(a, n) = a + n = \Theta(a).$$

The number of steps required by the process is the number of internal vertices of the tree and grows as  $\Theta(a^n)$  with  $a$  and  $n$ :

$$R(a, 0) = 1, \\ R(a, n) = 1 + \lceil a/d(n) \rceil + \sum_{i=0}^{\lceil a/d(n) \rceil - 1} R(a - i \times d(n), n - 1) = \Theta(a^n).$$

A mathematically rigorous and very clear solution is given by **Sébastien Gignoux** at his SICP Solutions website. The process is linear in space. It is  $O(a^n)$  in time, where  $a$  = amount and  $n$  = number of denominations of coins. Any comments below to the contrary are inaccurate. **BrianHagerty**

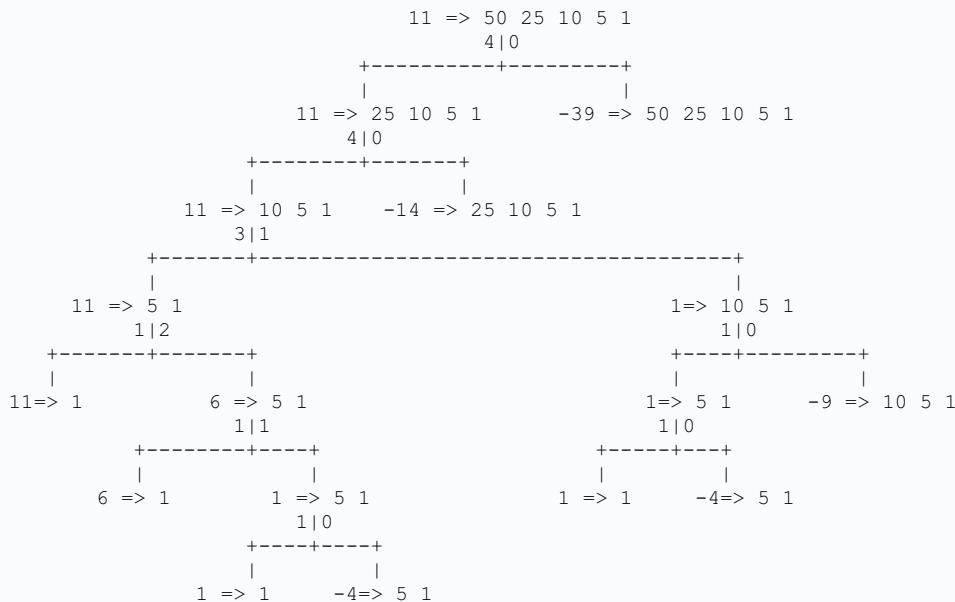
WARNING: one reader found this analysis to be highly inaccurate. For a correct one, see: (dead link: [www.ysagade.nl/2015/04/12/sicp-change-growth/](http://www.ysagade.nl/2015/04/12/sicp-change-growth/))

Solution to Exercise 1.14. Draw the tree illustrating the process generated by the count-change procedure of section 1.2.2 in making change for 11 cents. What are the orders of growth of the space and number of steps used by this process as the amount to be changed increases?

The tree illustration can be as given below. The order of value of coins have been reversed in order to reduce the size of the diagram. As said in the section 1.2.2, order of coins does not matter. So the resultant procedure will be as given below.

```
(define (count-change amount)
  (cc amount 5))
(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
        (else (+ (cc amount
                      (- kinds-of-coins 1))
                  (cc (- amount
                            (first-denomination kinds-of-coins))
                      kinds-of-coins)))))

(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 50)
        ((= kinds-of-coins 2) 25)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 5)
        ((= kinds-of-coins 5) 1)))
```



Per Rspns777: The analysis below is wrong.

TLDR;
Time Complexity: $O(n^5)$
Space Complexity: $O(n)$

The exercise also asks for the orders of growth in space and time, as the amount increases. (Forgive any errors or imprecision in my answer, as my analysis skills are still very basic.)

Space required (maximum height of the call tree) grows linearly with the amount, as it is determined by the number of times the smallest denomination divides into the amount. i.e.  $O(a)$

Time required (number of operations) grows in relation to  $O(a^n)$ . i.e.  $O(a * a * a * ...)$  since the 2nd branch is  $O(a)$ , and the first branch is called  $O(n)$  times.

I made a (dead link: [telegraphics.com.au/~toby/sicp/ex1-14.svg](http://telegraphics.com.au/~toby/sicp/ex1-14.svg) pretty svg version) of the call tree for 11 cents, generated by a slightly **modified version of the function** writing a Graphviz description.

**qu1j0t3**

I think space is  $O(n^2)$  as this is not a tail recursive function.

---

No, space for this recursive version is  $O(n)$ . The only space requirement is remembering the stack, and stack depth increases with depth of the tree.

For a true tail-recursive function, there is only ever one function in the stack, so space would be  $O(1)$ .

samphilipd?

---

Another more formal approach:

count-change space order of growth:

- $O(n)$  because max depth is  $n$

count-change time order of growth:

1.  $cc(n, 1) = O(n)$
2.  $cc(n, 2) = cc(n, 1) + cc(n-5, 2)$
3. each 2. step is  $O(n)$  and there are roughly  $n/5$  such steps
4. so we have  $O(n^2)$
5. by analogue we get  $O(n^k)$  ( $k$  currencies) for  $cc(n, k)$

Feel free to expand if needed.

**gollum0**

---

Slightly more detailed analysis is [here](#). The wonderful explanation below has wrong space asymptotics.

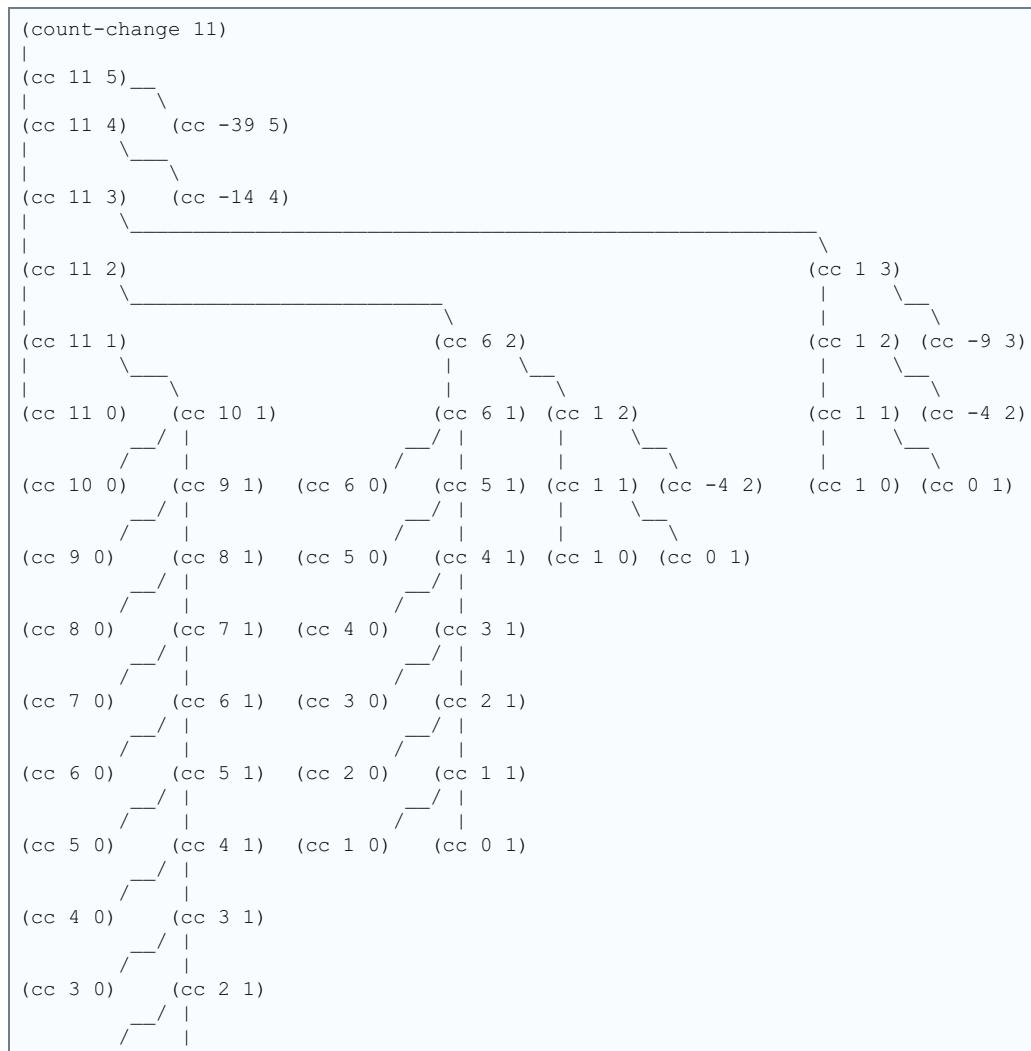
voom4000

---

### Another wonderful explanation

---

For fun, here is an ASCII graph if we keep the denominations in their original order:



```
(cc 2 0)      (cc 1 1)
   /   |
 (cc 1 0)     (cc 0 1)
```

---

Last modified : 2022-11-13 19:10:30  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-1.15

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

<< Previous exercise (1.14) | sicp-solutions | Next exercise (1.16) >>

---

The expression `(sine 12.15)` will be expanded by the interpreter as follows.

```
(sine 12.15)
(p (sine 4.05))
(p (p (sine 1.35)))
(p (p (p (sine 0.45))))
(p (p (p (p (sine 0.15)))))
(p (p (p (p (p (sine 0.05))))))
(p (p (p (p (p 0.05)))))
```

We can see that the procedure `p` is applied five times.

The angle `a` is divided by 3 each time the procedure `p` is applied. Expressing this differently, we can say that `p` is applied once for each complete power of 3 contained within the angle `a`. Therefore, given a positive argument, we can compute the number of times `p` is applied as the ceiling of the base 3 logarithm of the argument divided by 0.1, or `(ceiling(/ (log (/ 12.15 0.1)) (log 3)))`.

If we measure the required space and the number of steps by counting the invocations of `p`, the order of growth of the process generated by `(sine a)` is logarithmic. Exactly, the number of steps required are `(ceiling(/ (log (/ a 0.1)) (log 3)))`.

In other words we have  $O(\log(a))$  order of growth.

---

Last modified : 2021-05-08 17:24:43  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**

# sicp-ex-1.16

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (1.15) | sicp-solutions | Next exercise (1.17) >>

## Solution 1

```
(define (iter-fast-expt b n)
  (define (iter N B A)
    (cond ((= 0 N) A)
          ((even? N) (iter (/ N 2) (square B) A))
          (else (iter (- N 1) B (* B A))))))
  (iter n b 1))
```

Claim.

```
(iter N B A) = B^N * A
```

This claim implies that iter-fast-expt is correct:

```
(iter-fast-expt b n)
= (iter n b 1)
= b^n * 1 by claim
= b^n
```

Proof of claim.

Induction on N.

Case 1: N = 0.

```
(iter N B A)
= (iter 0 B A)
= A
= B^0 * A
= B^N * A
```

Case 2: N > 0 and N is even.

```
(iter N B A)
= (iter N/2 B^2 A)
= (B^2)^(N/2) * A by the inductive hypothesis
= B^N * A
```

Case 3: N > 0 and N is odd.

```
(iter N B A)
= (iter N-1 B B*A)
= B^(N-1) * B*A by the inductive hypothesis
= B^N * A
```

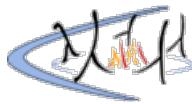
QED.

## Solution 2

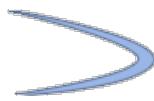
```
(define (fast-expt b n)
  (define (cube x) (* x x x))
  (define (fast-expt-iter b a counter)
    (cond ((= counter 0) a)
          ((= counter 1) (* a b))
          ((even? counter) (fast-expt-iter
                            (square b)
                            (* (square b) a)
                            (- (/ counter 2) 1)))
          (else (fast-expt-iter
                  (square b)
```

```
(* (cube b) a)
(- (/ (- counter 1) 2) 1))))))
(fast-expt-iter b 1 n))
```

Last modified : 2020-02-07 12:20:21  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-1.17



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (1.16) | sicp-solutions | Next exercise (1.18) >>

An example for illustrating the algorithm:  $3 * 10 = 3 * (2 * (10 / 2)) = 3 * (2 * 5) = 6 * 5 = 6 * (4 + 1) = 6 * 4 + 6 = 6 * (2 * (4 / 2)) + 6 = 6 * (2 * 2) + 6 = 12 * 2 + 6 = 24 + 6 = 30$ .

```
;; ex 1.17

;; Assume double and halve are defined by the language
(define (double x) (+ x x))
(define (halve x) (/ x 2))

(define (* a b)
  (cond ((= b 0) 0)
        ((even? b) (double (* a (halve b))))
        (else (+ a (* a (- b 1))))))

;; Testing
(* 2 4)
(* 4 0)
(* 5 1)
(* 7 10)
```

Here is an illustration of the recursive process:

```
; (fast-mult 3 7)
; (+ 3 (fast-mult 3 6))
; (+ 3 (double (fast-mult 3 3)))
; (+ 3 (double (+ 3 (fast-mult 3 2))))
; (+ 3 (double (+ 3 (double (fast-mult 3 1)))))
; (+ 3 (double (+ 3 (double (+ 3 (fast-mult 3 0))))))
; (+ 3 (double (+ 3 (double (+ 3 0)))))
; (+ 3 (double (+ 3 (double 3))))
; (+ 3 (double (+ 3 6)))
; (+ 3 18)
; 21
```

using tail recursion achieves better space order of growth of theta(1)

```
(define (fast-mult-by-add a b)
  (define (double x) (+ x x))
  (define (halve x) (/ x 2))

  (define (helper a b product) ;; "add a" b times
    (cond ((= b 0) product)
          ((even? b) (helper (double a) (halve b) product))
          (else (helper a (- b 1) (+ a product)))))

  (helper a b 0)))
```

bob

Isn't this version just skipping ahead to the next one, Exercise 1.18?

ybsh

I agree with Bob. The problem instruction requires readers to implement a procedure analogous to fast-expt (from 1.2.4), which evolves into a recursive process, not an iterative one. Anyway, nice explanation.

master

Doesn't this accomplish the same thing but more efficiently? No need to delay the doubling.

```
(define (double x) (+ x x))
```

```
(define (halve x) (/ x 2))

(define (fast-mult a b)
  (cond ((= b 0) 0)
        ((even? b) (fast-mult (double a) (halve b)))
        (else (+ a (fast-mult a (- b 1))))))
```

bidouille

I also made the master way. Not sure it changes about efficiency, but at least I find it more readable :)

jazbot

I came up with the following solution, which seems to execute fewer calls than the solutions listed above:

```
(define (fast-* a b)
  (cond ((= b 1)
         a)
        ((even? b)
         (double (fast-* a (halve b))))
        (else
         (+ a (double (fast-* a (halve (- b 1))))))))
```

For example if I calculate the product of 3 and 10,000,000, my solution uses 24 calls, while both of the solutions above use 32.

anon

I like the modification by jazbot to call the procedure once instead of twice for odd b but i believe the procedure would throw an error for b=0. For most use cases, the condition of b=1 catches the reduction of b through the recursive process but if you start with b=0, then you should run into an issue.

Easily solved by adding in another condition though.

# sicp-ex-1.18

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (1.17) | sicp-solutions | Next exercise (1.19) >>

The solution presented here is based on the solutions for [sicp-ex-1.16](#) and [sicp-ex-1.17](#).

```
; ; ex 1.18. Based on exercises 1.16 and 1.17

; ; Assume double and halve are defined by the language
(define (double x) (+ x x))
(define (halve x) (floor (/ x 2)))

(define (* a b)
  (define (iter accumulator a b)
    (cond ((= b 0) accumulator)
          ((even? b) (iter accumulator (double a) (halve b)))
          (else (iter (+ accumulator a) a (- b 1)))))
  (iter 0 a b))

; ; Testing
(* 2 4)
(* 4 0)
(* 5 1)
(* 7 10)

; ; Alternate version, which makes more complete use of the
; ; Russian Peasant Algorithm in footnote 40.  Uses roughly half
; ; the steps of the above
(define (double a) (+ a a))
(define (halve a) (/ a 2))

(define (mult3 a b)
  (define (mult-iter accumulator b c)
    (cond ((= c 0) accumulator)
          ((even? c) (mult-iter accumulator (double b) (halve c)))
          (else (mult-iter (+ accumulator b) (double b) (- (halve c) 0.5))))))
  (mult-iter 0 a b))

; ; Testing
(mult3 2 4)
(mult3 4 0)
(mult3 5 1)
(mult3 7 10)
```

# sicp-ex-1.19

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (1.18) | sicp-solutions | Next exercise (1.20) >>

This exercise is fairly easy if we observe that, if we group  $a$  and  $b$  into a column vector,  $T$  is a linear transformation expressed by

1	1
1	0

and  $T_{pq}$  is, similarly,

p+q	q
q	p

Given this definition of  $T_{pq}$ ,  $T_p \cdot q'$  can be easily computed as the square of  $T_{pq}$ .  $p'$  and  $q'$  are, respectively

$$\begin{aligned} p' &= p^2 + q^2 \\ q' &= 2pq + q^2 \end{aligned}$$

The same results can be computed without resorting to linear algebra. Just define  $a'$  and  $b'$  by applying the transformation once

$$\begin{aligned} a' &= qb + qa + pa = (p + q)a + bq \\ b' &= qa + pb \end{aligned}$$

Then, let  $a''$  and  $b''$  be the results of applying the transformation to  $a'$  and  $b'$  and show how those values can be computed directly from  $a$  and  $b$

$$\begin{aligned} a'' &= qb' + qa' + pa' = (p + q)a' + b'q = \dots = (p^2 + 2pq + 2q^2)a + (2pq + q^2)b \\ b'' &= qa' + pb' = \dots = (2pq + q^2)a + (p^2 + q^2)b \end{aligned}$$

Now, it is plain to see that  $p'$  and  $q'$  are the same as the ones shown above.

Putting this all together, we can complete the given procedure

```
;; ex 1.19

(define (fib n)
  (fib-iter 1 0 0 1 n))
(define (fib-iter a b p q count)
  (cond ((= count 0) b)
        ((even? count)
         (fib-iter a
                   b
                   (+ (square p) (square q))
                   (+ (* 2 p q) (square q))
                   (/ count 2)))
        (else (fib-iter (+ (* b q) (* a q) (* a p))
                        (+ (* b p) (* a q)))
                       p
                       q
                       (- count 1)))))

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

;; Testing
(fib 0)
(fib 1)
(fib 2)
(fib 3)
(fib 4)
(fib 5)
(fib 6)
(fib 7)
(fib 8)
(fib 9)
```

(fib 10)

These are my self-notes as I worked through the problem. My goal was not just to implement the missing parts of the procedure (I didn't look at the book's implementation until after), but to create the entire procedure from scratch along with fully understanding the thought process behind it.

Jordan Chavez

```
#|
Consider the fibonacci numbers as the result of applying the following transformation:
T(a, b) = (a + b, a)
a <- a + b
b <- a
Starting with (a, b) = (0, 1) as first input and applied n times.

This can be thought of as a special case of the following transformation Tpq, defined below, with
p = 0 and q = 1:

Tpq(a, b) = (bq + aq + ap, bp + aq)
a <- (bq + aq + ap)
b <- (bp + aq)

T01(a, b) = (b1 + a1 + a0, b0 + a1) = (a + b, a)

Now we need to calculate how to successively apply a particular Tpq twice. In other words, given
Tpq and Txy, compute p' and q' such that Tp'q'(a, b) = Tpq(Txy(a, b))

Tpq(Txy(a, b))
Tpq(by + ay + ax, bx + ay)
((bx + ay)q + (by + ay + ax)q + (by + ay + ax)p, (bx + ay)p + (by + ay + ax)q)

Calculation for a <- (bq + aq + ap):
(bx + ay)q + (by + ay + ax)q + (by + ay + ax)p
bxq + ayz + byq + ayz + axq + bxp + ayp + axp
b(xq + yp + yq) + a(xq + yp + yq) + a(xp + yq)

Calculation for b <- (bp + aq):
(bx + ay)p + (by + ay + ax)q
bxp + ayp + byq + ayz + axq
b(xp + yq) + a(xq + yp + yq)

Both calculations confirm:
p' = xp + yq
q' = xq + yp + yq

For the special case of x = p and y = q, we get:
p' = pp + qq
q' = 2pq + qq

Also note that T is commutative. Swap p<->x and q<->y and you get the same result:
p' = px + qy
q' = py + qx + qy

Finally, calculate pi and qi, the identity values such that Tpiqi(Txy) = Txy:
x = xpi + yqi
yq' = xqi + ypi + yqi
clearly (pi, qi) = (1, 0)
These identity values will form the first parameters of our "accumulator" T.

To compute fib(n), apply T01 n times to (1, 0), and take the *second* value, i.e. b

Now we can implement a logarithmic fibonacci procedure using this idea.

To understand this, think of the exponentiation example where we did a logarithmic+iterative
implementation. We had an accumulator for the "odd" factors and we successively doubled the base
while halving the exponent.
An example is a^15. Each number in the text below represents a power of a, so 1 means a^1, 2
means a^2, etc:

accumulator is on the left, rest is on the right:
0 11111111111111 (acc = 1,      base = a^1, exp = 15)
1 11111111111111 (acc = a,      base = a^1, exp = 14)
1 2222222222222222 (acc = a,      base = a^2, exp = 7)
12 2222222222222222 (acc = a^3,    base = a^2, exp = 6)
12 4444444444444444 (acc = a^3,    base = a^4, exp = 3)
124 4444444444444444 (acc = a^7,    base = a^4, exp = 2)
1248 4444444444444444 (acc = a^7,    base = a^8, exp = 1)
1248 (acc = a^15,   base = a^8, exp = 0)

We can apply the same idea.
Each time you replace Tpq with Tp'q', you get to halve the number of times you apply it (halving
the exponent), since you're doubling the "power" of each application (squaring the base).
In other words T^n = (T^2) ^ (n/2)
```

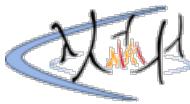
Our "accumulator" is simply p-acc and q-acc that represent the parameters for the accumulated T function. Our "base" is p and q that change as we "double" the base function.  
At the end we'll have p-acc and q-acc equivalent to applying T01 n times. We can compute the actual fibonacci number by taking our p-acc and q-acc, applying them to (1, 0), and taking the second value:

```
a <- (bq + aq + ap) = (0q + 1q + 1p) = p + q    <-- we don't care about this one at the end (it's fib(n+1))
b <- (bp + aq) = (0p + 1q) = q      <--- this is the final value fib(n)

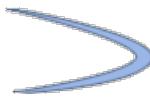
| #

(define (fib n)
  (define (even? x)
    (= (remainder x 2) 0))
  (define (fib-iter p-acc q-acc p q n)
    (cond ((= n 0) q-acc)
          ((even? n) (fib-iter
                        p-acc
                        q-acc
                        (+ (* p p) (* q q))
                        (+ (* 2 p q) (* q q))
                        (/ n 2)))
          (else (fib-iter
                    (+ (* p p-acc) (* q q-acc))
                    (+ (* p q-acc) (* q p-acc) (* q q-acc))
                    p
                    q
                    (- n 1))))))
  (fib-iter 1 0 0 1 n))

(fib 0)
(fib 1)
(fib 2)
(fib 5)
(fib 6)
(fib 19)
(fib 20)
```



# sicp-ex-1.20



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (1.19) | sicp-solutions | Next exercise (1.21) >>

The process generated using the normal-order evaluation is the following. It performs  $18_{\text{remainder}}$  operations: 14 when evaluating the condition and 4 in the final reduction phase.

```
(gcd 206 40)
(if (= 40 0) ...)
(gcd 40 (remainder 206 40))
(if (= (remainder 206 40) 0) ...)
(if (= 6 0) ...)
(gcd (remainder 206 40) (remainder 40 (remainder 206 40)))
(if (= (remainder 40 (remainder 206 40)) 0) ...)
(if (= 4 0) ...)
(gcd (remainder 40 (remainder 206 40)) (remainder (remainder 206 40) (remainder 40 (remainder 206 40))))
(if (= (remainder (remainder 206 40) (remainder 40 (remainder 206 40))) 0) ...)
(if (= 2 0) ...)
(gcd (remainder (remainder 206 40) (remainder 40 (remainder 206 40))) (remainder (remainder 206 40) (remainder 40 (remainder 206 40))))
(if (= (remainder (remainder 40 (remainder 206 40)) (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))) 0) ...)
(if (= 0 0) ...)
(remainder (remainder 206 40) (remainder 40 (remainder 206 40)))
```

The number 'R' of  $\text{remainder}$  operations executed by the normal-order evaluation process is given by the formula

$$R = \sum_{i=1}^n (\text{fib}(i) + \text{fib}(i-1)) - 1$$

where  $n$  is the number of  $\text{gcd}$  invocations required to compute the  $(\text{gcd } a \ b)$ . The first invocation does not need to compute  $\text{remainder}$  thus the  $-1$

$R$  is given by the following function:

```
(define (count-remainders n)
  (define (loop n sum)
    (if (= 0 n) (- sum 1)
        (loop (- n 1) (+ sum (fib n) (fib (- n 1))))))
  (loop n 0))
```

In this particular case that is:

```
(count-remainders 5)
=> 18
```

The process generated using the applicative-order evaluation is the following. It performs  $4_{\text{remainder}}$  operations.

```
(gcd 206 40)
(gcd 40 (remainder 206 40))
```

```
(gcd 40 6)
(gcd 6 (remainder 40 6))
(gcd 6 4)
(gcd 4 (remainder 6 4))
(gcd 4 2)
(gcd 2 (remainder 4 2))
(gcd 2 0)
2
```

It seems that the formula

$$R = \text{SUM}(i \text{ from } 1 \text{ to } n, \text{fib}(i) + \text{fib}(i - 1)) - 1$$

is incorrect. One can easily check this by letting  $n=2$ . The correct count should be 1 while the formula gives 2.

Let  $b_i$  be the value of  $b$  at the  $n$ 'th invocation of  $\text{gcd}(a,b)$ . Let  $b(i)$  be the count of remainder procedure needed to calculate  $b_i$ . Similarly, define  $a_i$  and  $a(i)$ . It is easy to check that

1.  $a_i = b_{i-1}$  and thus  $a(i) = b(i-1)$ .
2.  $b(i+1) = a(i) + b(i) + 1 = b(i-1) + b(i) + 1$  because  $b_{i+1} = a_i \bmod b_i$

Based on my own derivation,  $R$  should be

$b(1) + b(2) + \dots + b(n) + b(n-1)$  with  $b(1)=0$ ,  $b(2)=1$  and  $b(n)=b(n-1)+b(n-2)+1$ , which is not equivalent with the above  $R$  formula

The correct formula is

$$R(n) = \text{SUM}(i \text{ from } 1 \text{ to } n - 1, \text{fib}(i)) + \text{fib}(n - 2) - 1$$

Where  $n$  is the number of applications.

$$R(2) = 1$$

$$R(3) = 4$$

$$R(5) = 17$$

The above formulas are wrong. The truly correct one should be:

$$R(n) = \text{SUM}(i \text{ from } 1 \text{ to } n, \text{fib}(i+1) - 1) + \text{fib}(n) - 1$$

where  $n$  is the number of times  $\text{gcd}(a, b)$  is called.

We have,

$$R(1) = 0$$

$$R(2) = 1$$

$$R(3) = 4$$

$$R(4) = 9$$

$$R(5) = 18$$

Below is the derivation:

Denote  $n$  as the number of times  $\text{gcd}()$  is called,  $R(n)$  the number of times  $\text{remainder}()$  is invoked.

For  $(\text{gcd } a(k) \ b(k))$ , let  $\text{num\_a}(k)$  be the number of  $\text{remainder}()$  in  $a(k)$ ,  $\text{num\_b}(k)$  be the number of  $\text{remainder}()$  in  $b(k)$ , where  $k=1,2,\dots,n$ .

Then we have the updating process:

$$a(k+1) = b(k)$$

$$b(k+1) = \text{remainder}(a(k), b(k))$$

Thus,

$\text{num\_a}(1) = 0$

$\text{num\_b}(1) = 0$

$\text{num\_a}(k+1) = \text{num\_b}(k)$

$R(k+1) = \text{num\_a}(k) + \text{num\_b}(k) + 1 = \text{num\_b}(k+1)$

By substituting  $\text{num\_a}$  for  $\text{num\_b}$ , we get the following for  $\{\text{num\_b}(k)\}$ ,  $k=1,2,\dots,n$ ,

$\text{num\_b}(1) = 0$

$\text{num\_b}(2) = 1$

$\text{num\_b}(k+1) = \text{num\_b}(k) + \text{num\_b}(k-1) + 1$

Obviously we could get

$\text{num\_b}(k) = \text{fib}(k+1) - 1$

(hello Lily.X, the above step is not obvious for me, can you explain?)

Since

```
R(n) = SUM(i from 1 to n, num_b(i)) + num_a(n)
```

which is quite obvious following the normal-order-evaluation rule for if statement, we could get

```
R(n) = SUM(i from 1 to n, fib(i+1)- 1) + fib(n) -1
```

:) Lily.X

The correct formula above can also be written as

```
R(n) = 2*fib(n+2)-n-3
```

# sicp-ex-1.21

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

<< Previous exercise (1.20) | sicp-solutions | Next exercise (1.22) >>

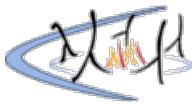
---

The smallest divisor of each of 199, 1999, 19999 is, respectively, 199, 1999, 7.

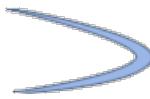
```
> (smallest-divisor 199)
199
> (smallest-divisor 1999)
1999
> (smallest-divisor 19999)
7
```

---

Last modified : 2017-11-09 14:56:48  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-1.22



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (1.21) | sicp-solutions | Next exercise (1.23) >>

```
;; ex 1.22

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

(define (smallest-divisor n)
  (find-divisor n 2))

(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))

(define (divides? a b)
  (= (remainder b a) 0))

(define (prime? n)
  (= n (smallest-divisor n)))

(define (timed-prime-test n)
  (start-prime-test n (runtime)))

(define (start-prime-test n start-time)
  (if (prime? n)
      (report-prime n (- (runtime) start-time)))))

(define (report-prime n elapsed-time)
  (newline)
  (display n)
  (display " *** ")
  (display elapsed-time))

(define (search-for-primes lower upper)
  (define (iter n)
    (cond ((<= n upper) (timed-prime-test n) (iter (+ n 2))))
          (iter (if (odd? lower) lower (+ lower 1)))))

(search-for-primes 1000 1019) ; 1e3
(search-for-primes 10000 10037) ; 1e4
(search-for-primes 100000 100043) ; 1e5
(search-for-primes 1000000 1000037) ; 1e6

; As of 2008, computers have become too fast to appreciate the time
; required to test the primality of such small numbers.
; To get meaningful results, we should perform the test with numbers
; greater by, say, a factor 1e6.
(newline)
(search-for-primes 1000000000 1000000021) ; 1e9
(search-for-primes 10000000000 10000000061) ; 1e10
(search-for-primes 100000000000 100000000057) ; 1e11
(search-for-primes 1000000000000 100000000063) ; 1e12
```

The output produced by the above code is:

```
1009 *** 0.
1013 *** 0.
1019 *** 0.
10007 *** 0.
10009 *** 0.
10037 *** 0.
100003 *** 0.
100019 *** 0.
100043 *** 0.
1000003 *** 0.
1000033 *** 0.
1000037 *** 1.0000000000000002e-2

100000007 *** .13
100000009 *** .11999999999999997
```

```

10000000021 *** .10999999999999999
10000000019 *** .39
10000000033 *** .3799999999999999
10000000061 *** .3700000000000001
100000000003 *** 1.22
100000000019 *** 1.2300000000000004
100000000057 *** 1.2199999999999998
100000000039 *** 3.829999999999999
100000000061 *** 4.039999999999999
100000000063 *** 3.9700000000000006

```

From our timing data, we can observe that, when increasing the tested number of a factor 10, the required time increases roughly of a factor 3. By noting that  $3 \approx \sqrt{10}$ , we can confirm both the growth prediction and the notion that programs run in a time proportional to the steps required for the computation.

## Notes

1. This exercise requires a Scheme implementations which provides a `runtime` primitive, such as **MIT/GNU Scheme** or `lang sicp` for DrRacket.
2. The arguments to `(search-for-primes)` are an integer range. Therefore, to find the first three primes above a certain value, I needed to run the program with an arbitrary upper bound until three primes were found, and then put the third prime as the largest value.
3. In order to get a cleaner output, I modified `timed-prime-test` to make it print only prime numbers, not each tested number.
4. By using the `begin` construct, not introduced yet in the book, the inner procedure can be rewritten without repeating the `if` test:

```

(define (search-iter cur last)
  (if (<= cur last)
      (begin (timed-prime-test cur)
             (search-iter (+ cur 2) last))))

```

Another implementation also easy to understand:

```

;basic operations
(define (square x)
  (* x x))
(define (divides? a b)
  (= (remainder b a) 0))
(define (even? n)
  (= (remainder n 2) 0))
;smallest divisor computation
(define (smallest-divisor n)
  (define (find-divisor n test)
    (cond ((> (square test) n) n)
          ((divides? test n) test)
          (else (find-divisor n (+ test 1)))))

  (find-divisor n 2)))
;primality check
(define (prime? n)
  (= n (smallest-divisor n)))
;check primality of consecutive odd integers in some range
;time&primality test
;drRacket has no (runtime) variable; had to substitute it with (current-milliseconds) which is basically same
(define (runtime)
  (current-milliseconds))
(define (timed-prime-test n)
  (start-prime-test n (runtime)))
(define (start-prime-test n start-time)
  (if (prime? n)
      (report-prime n (- (runtime) start-time))
      #f))
(define (report-prime n elapsed-time)
  (display n)
  (display "***")
  (display elapsed-time)
  (newline))
;search counter
(define (search-for-primes n counter)
  (if (even? n)
      (s-f-p (+ n 1) counter)
      (s-f-p n counter)))
;it's important to pay attention to the fact that predicate of the first 'if' here calls (timed-prime-test n) which in case of #t computes into two procedures - (report-prime n (elapsed-time)) and 'then' case of the first 'if'.
(define (s-f-p n counter)
  (if (> counter 0)
      (if (timed-prime-test n)
          (s-f-p (+ n 2) (- counter 1))
          (s-f-p (+ n 2) counter))
      "COMPUTATION COMPLETE"))

```

**Output:**

```
(search-for-primes 1000 3)
1009***0
1013***0
1019***0
"COMPUTATION COMPLETE"
(search-for-primes 1000000 3)
1000003***0
1000033***0
1000037***0
"COMPUTATION COMPLETE"
(search-for-primes 1000000000000 3)
100000000039***359
100000000061***359
100000000063***406
"COMPUTATION COMPLETE"
```

---

Last modified : 2020-02-11 15:11:42  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**

# sicp-ex-1.23

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (1.22) | sicp-solutions | Next exercise (1.24) >>

This exercise is based on [sicp-ex-1.22](#). Please read the notes therein.

```
;; ex 1.23

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

(define (smallest-divisor n)
  (find-divisor n 2))

(define (find-divisor n test-divisor)
  (define (next n)
    (if (= n 2) 3 (+ n 2)))
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (next test-divisor)))))

(define (divides? a b)
  (= (remainder b a) 0))

(define (prime? n)
  (= n (smallest-divisor n)))

(define (timed-prime-test n)
  (start-prime-test n (runtime)))

(define (start-prime-test n start-time)
  (if (prime? n)
      (report-prime n (- (runtime) start-time)))))

(define (report-prime n elapsed-time)
  (newline)
  (display n)
  (display "***")
  (display elapsed-time))

(timed-prime-test 1009)
(timed-prime-test 1013)
(timed-prime-test 1019)
(timed-prime-test 10007)
(timed-prime-test 10009)
(timed-prime-test 10037)
(timed-prime-test 100003)
(timed-prime-test 100019)
(timed-prime-test 100043)
(timed-prime-test 1000003)
(timed-prime-test 1000033)
(timed-prime-test 1000037)

; See comments in exercise 1.22
(newline)
(timed-prime-test 1000000007)
(timed-prime-test 1000000009)
(timed-prime-test 1000000021)
(timed-prime-test 1000000019)
(timed-prime-test 1000000033)
(timed-prime-test 1000000061)
(timed-prime-test 1000000003)
(timed-prime-test 10000000019)
(timed-prime-test 10000000057)
(timed-prime-test 100000000039)
(timed-prime-test 100000000061)
(timed-prime-test 100000000063)
```

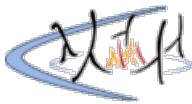
The output produced by the above code is:

```
1009 *** 0.
1013 *** 0.
1019 *** .01
```

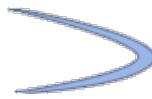
```
10007 *** 0.  
10009 *** 0.  
10037 *** 0.  
100003 *** 0.  
100019 *** 0.  
100043 *** 0.  
1000003 *** 0.  
1000033 *** .01  
1000037 *** 0.  
  
1000000007 *** .08  
1000000009 *** .09  
1000000021 *** .08000000000000000002  
10000000019 *** .25  
10000000033 *** .26  
10000000061 *** .27  
10000000003 *** .8299999999999998  
10000000019 *** .8200000000000003  
100000000057 *** .7999999999999998  
100000000039 *** 2.6500000000000004  
100000000061 *** 2.59  
100000000063 *** 2.59
```

The observed ratio of the speed of the two algorithms is not 2, but roughly 1.5 (or 3:2).

This is mainly due to the NEXT procedure's IF test. The input did halve indeed, but we need to do an extra IF test.



# sicp-ex-1.24



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (1.23) | sicp-solutions | Next exercise (1.25) >>

This exercise is based on [sicp-ex-1.22](#). Please read the notes therein.

```
;; ex 1.24

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

(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (square (expmod base (/ exp 2) m))
                    m))
        (else
         (remainder (* base (expmod base (- exp 1) m))
                    m)))))

(define (fermat-test n)
  (define (try-it a)
    (= (expmod a n n) a))
  (try-it (+ 1 (random (- n 1)))))

(define (fast-prime? n times)
  (cond ((= times 0) true)
        ((fermat-test n) (fast-prime? n (- times 1)))
        (else false)))

(define (prime? n)
  ; Perform the test how many times?
  ; Use 100 as an arbitrary value.
  (fast-prime? n 100))

(define (timed-prime-test n)
  (start-prime-test n (runtime)))

(define (start-prime-test n start-time)
  (if (prime? n)
      (report-prime n (- (runtime) start-time)))))

(define (report-prime n elapsed-time)
  (newline)
  (display n)
  (display " *** ")
  (display elapsed-time))

(timed-prime-test 1009)
(timed-prime-test 1013)
(timed-prime-test 1019)
(timed-prime-test 10007)
(timed-prime-test 10009)
(timed-prime-test 10037)
(timed-prime-test 100003)
(timed-prime-test 100019)
(timed-prime-test 100043)
(timed-prime-test 1000003)
(timed-prime-test 1000033)
(timed-prime-test 1000037)

; See comments in exercise 1.22
(newline)
(timed-prime-test 1000000007)
(timed-prime-test 1000000009)
(timed-prime-test 1000000021)
(timed-prime-test 1000000019)
(timed-prime-test 10000000033)
(timed-prime-test 10000000061)
(timed-prime-test 100000000003)
(timed-prime-test 100000000019)
(timed-prime-test 100000000057)
(timed-prime-test 100000000039)
(timed-prime-test 100000000061)
(timed-prime-test 100000000063)
```

The output produced by the above code is:

```
1009 *** .01
1013 *** 0.
1019 *** 9.99999999999998e-3
10007 *** 0.
10009 *** 1.0000000000000002e-2
10037 *** 1.0000000000000002e-2
100003 *** 9.99999999999995e-3
100019 *** 1.0000000000000009e-2
100043 *** 9.99999999999995e-3
1000003 *** 9.99999999999995e-3
1000033 *** 1.0000000000000009e-2
1000037 *** .0199999999999999
1000000007 *** 1.0000000000000009e-2
1000000009 *** .0199999999999999
1000000021 *** 2.000000000000018e-2
1000000019 *** .0199999999999999
1000000033 *** 1.0000000000000009e-2
1000000061 *** .0199999999999999
10000000003 *** .03
1000000019 *** 2.000000000000018e-2
10000000057 *** 1.999999999999962e-2
10000000039 *** 3.000000000000027e-2
10000000061 *** 2.000000000000018e-2
10000000063 *** .0299999999999999
```

From the collected timing data, we can observe that testing a number with twice as many digits (1e12 vs. 1e6) is roughly twice as slow. This supports the theory of logarithmic growth.

[torinmr]: I tested this using much larger numbers (50-200 digits) and obtained slightly different results:

```
45 digits => 0.04 seconds
90 digits => 0.1 seconds
135 digits => 0.18 seconds
180 digits => 0.27 seconds
```

If the function ran in logarithmic time, we would expect running time to be linear in the number of digits, but the growth is faster than that. This is probably because performing primitive operations on sufficiently large numbers is not constant time, but grows with the size of the number.

AnScímeoirBeag

There are limits on the `random` procedure in some Scheme implementations (e.g. Racket) preventing fast-prime from reaching values large enough to show non-zero timings on modern computers.

To avoid this in DrRacket add: `(#%require (lib "27.ss" "srfi"))` to gain access to the `random-integer` procedure.

Truly massive numbers are needed on modern machines (e.g 10<sup>156</sup> and 10<sup>312</sup> in my case). My observed ratios were 2.7-3.4, most likely because operations on large integers (above the normal 32/64-bit limit) are not constant in time, but functions of the number's size\*, as mentioned by torinmr.

\*This is due to larger numbers being stored as a n-tuple of 32/64-bit numbers, with n increasing as they grow in size.

tiendo1011

Since fermat-test uses a random number, it's possible that the number we use when testing 1,000,000 is not 1000 times larger than the number we use when testing 1000. Hence the difference in runtime.

# sicp-ex-1.25

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (1.24) | sicp-solutions | Next exercise (1.26) >>

The modified version of `expmod` computes huge intermediate results.

Scheme is able to handle arbitrary-precision arithmetic, but arithmetic with arbitrarily long numbers is computationally expensive. This means that we get the same (correct) results, but it takes considerably longer.

For example:

```
(define (square m)
  (display "square ") (display m) (newline)
  (* m m))

=> (expmod 5 101 101)
square 5
square 24
square 71
square 92
square 1
square 1
5
=> (remainder (fast-expt 5 101) 101)
square 5
square 25
square 625
square 390625
square 152587890625
square 23283064365386962890625
5
```

The `remainder` operation inside the original `expmod` implementation, keeps the numbers being squared less than the number tested for primality  $m$ . `fast-expt` however squares huge numbers of  $a^m$  size.

tiendo1011

I wrote some procedures for us to help testing the speed:

```
; Helper procedures
(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2)))))
        (else (* b (fast-expt b (- n 1)))))

(define (square x) (* x x))
(define (report-elapsed-time start-time)
  (display "***")
  (display (- (runtime) start-time)))

; The original & modified procedures
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder
          (square (expmod base (/ exp 2) m)) ; (1)
          m))
        (else
         (remainder
          (* base (expmod base (- exp 1) m))
          m)))))

(define (modified-expmod base exp m)
  (remainder (fast-expt base exp) m))

; Test the speed
(define start-time (runtime))
(expmod 999999 1000000 1000000)
(report-elapsed-time start-time)

(define start-time (runtime))
(modified-expmod 999999 1000000 1000000)
(report-elapsed-time start-time)
```

For the original implementation, the elapsed time taken is 543, while the modified version takes

1001921, which is significantly larger. The result maybe different on your machine, but I guess the ratio is somewhat similar.

# sicp-ex-1.26

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (1.25) | sicp-solutions | Next exercise (1.27) >>

Instead of a linear recursion, the rewritten `expmod` generates a tree recursion, whose execution time grows exponentially with the depth of the tree, which is the logarithm of  $N$ . Therefore, the execution time is linear with  $N$ .

tiendo1011

To simplify, we'll find the different between

`square(expmod base n m)` and `*` (`expmod base n m`) (`expmod base n m`)

1. `square(expmod base n m)` (Let's assume it takes  $a$  steps)

Double the size of the input:

`square(expmod base 2n m)` becomes `square(square(expmod base n m))`

Since `square(expmod base n m)` takes  $a$  steps, feed it to `square` takes  $a + 1$  steps

That's  $O(\log n)$  growth

2. `*` (`expmod base n m`) (`expmod base n m`) (Lets assume they take  $a$  steps)

Double the size of the input:

`*` (`expmod base 2n m`) (`expmod base 2n m`)

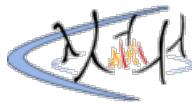
becomes `*` (`*` (`expmod base n m`) (`expmod base n m`)) (`*` (`expmod base n m`) (`expmod base n m`))

Each `*` (`expmod base n m`) (`expmod base n m`) takes  $a$  steps

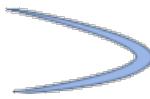
We have to calculate them twice so they take  $2a$  steps

That's  $O(n)$  growth

Last modified : 2020-08-22 04:05:46  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-1.27



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (1.26) | sicp-solutions | Next exercise (1.28) >>

To solve this exercise, we need to change fermat-test, making it test every integer number from 1 to  $n-1$ , not just a few random ones.

Then, we perform the test on Carmichael numbers and print whether the primality test has been fooled or not.

```
;; ex 1.27

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

(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (square (expmod base (/ exp 2) m))
                    m))
        (else
         (remainder (* base (expmod base (- exp 1) m))
                    m)))))

(define (full-fermat-prime? n)
  (define (iter a n)
    (if (= a n) true
        (if (= (expmod a n n) a) (iter (+ a 1) n) false)))
  (iter 1 n))

(define (test-fermat-prime n expected)
  (define (report-result n result expected)
    (newline)
    (display n)
    (display ": ")
    (display result)
    (display ": ")
    (display (if (eq? result expected) "OK" "FOOLED")))
  (report-result n (full-fermat-prime? n) expected))

(test-fermat-prime 2 true)
(test-fermat-prime 13 true)
(test-fermat-prime 14 false)

(test-fermat-prime 561 false) ; Carmichael number
(test-fermat-prime 1105 false) ; Carmichael number
(test-fermat-prime 1729 false) ; Carmichael number
(test-fermat-prime 2465 false) ; Carmichael number
(test-fermat-prime 2821 false) ; Carmichael number
(test-fermat-prime 6601 false) ; Carmichael number
```

An alternative with no nested ifs:

```
;; Parse error: Spurious closing paren found
(define (carmichel-test n)
  (define (iter a n)
    (cond ((= a n) #t)
          ((expmod a n n) a) (iter (+ a 1) n))
          (else #f)))
  (iter 1 n))

(carmichel-test 561)
(carmichel-test 1105)
(carmichel-test 1729)
(carmichel-test 2465)
(carmichel-test 2821)
(carmichel-test 6601)

;; non-carmichel numbers to test if it works
(carmichel-test 10)
(carmichel-test 155)
(carmichel-test 121)
```

master

Both of the above implementations fail to take into account the special case of 1 which by definition is not a prime number. Testing to see if n is equal to 1 solves this minor issue. Below is my solution to the exercise (because of the way the algorithm was implemented it is also necessary to test whether n is equal to 0 in order to avoid division by zero):

```
(define (car-test n)
  (define (car-test-iter n a)
    (cond ((or (= n 1) (= n 0)) false)
          ((= a n) true)
          ((not (= (expmod a n n) (remainder a n))) false)
          (else (car-test-iter n (+ a 1)))))

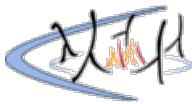
  (car-test-iter n 1))

(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder
          (square (expmod base (/ exp 2) m))
          m))
        (else
         (remainder
          (* base (expmod base (- exp 1) m))
          m)))))

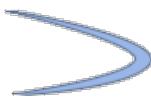
(define (square x) (* x x))

;; testing on Carmichael numbers
(car-test 561)
(car-test 1105)
(car-test 1729)
(car-test 2465)
(car-test 2821)
(car-test 6601)

;; testing on 1 and random numbers <100,000,000 (courtesy of random.org)
(car-test 1)
(car-test 85230658)
(car-test 13828192)
(car-test 69911818)
(car-test 49141805)
(car-test 29170450)
```



# sicp-ex-1.28



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (1.27) | sicp-solutions | Next exercise (1.29) >>

Maggyero

## QUESTION

One variant of the Fermat test that cannot be fooled is called the Miller–Rabin test (Miller 1976; Rabin 1980). This starts from an alternate form of Fermat’s Little Theorem, which states that if  $n$  is a prime number and  $a$  is any positive integer less than  $n$ , then  $a$  raised to the  $(n - 1)$ st power is congruent to 1 modulo  $n$ . To test the primality of a number  $n$  by the Miller–Rabin test, we pick a random number  $a < n$  and raise  $a$  to the  $(n - 1)$ st power modulo  $n$  using the `expmod` procedure. However, whenever we perform the squaring step in `expmod`, we check to see if we have discovered a “nontrivial square root of 1 modulo  $n$ ,” that is, a number not equal to 1 or  $n - 1$  whose square is equal to 1 modulo  $n$ . It is possible to prove that if such a nontrivial square root of 1 exists, then  $n$  is not prime. It is also possible to prove that if  $n$  is an odd number that is not prime, then, for at least half the numbers  $a < n$ , computing  $a^{(n - 1)}$  in this way will reveal a nontrivial square root of 1 modulo  $n$ . (This is why the Miller–Rabin test cannot be fooled.) Modify the `expmod` procedure to signal if it discovers a nontrivial square root of 1, and use this to implement the Miller–Rabin test with a procedure analogous to `fermat-test`. Check your procedure by testing various known primes and non-primes. Hint: One convenient way to make `expmod` signal is to have it return 0.

## ANSWER

```
(import (scheme small))
(import (srfi 27))

(define (miller-rabin-test n rounds)
  (define (try-it rounds)
    (define a (+ 1 (random-integer (- n 1))))
    (cond
      ((= rounds 0) #t)
      ((= (expmod a n n) a) (try-it (- rounds 1)))
      (else #f)))
  (define (expmod base e mod)
    (cond
      ((= e 0) 1)
      ((even? e) (squaremod (expmod base (/ e 2) mod) mod))
      (else (remainder (* base (expmod base (- e 1) mod)) mod))))
  (define (squaremod x mod)
    (define y (remainder (square x) mod))
    (if (and (= y 1) (not (= x 1)) (not (= x (- mod 1)))))
        0
        y))
  (and (> n 1) (try-it rounds)))

(display (miller-rabin-test 2 10)) (newline)
(display (miller-rabin-test 3 10)) (newline)
(display (miller-rabin-test 5 10)) (newline)
(display (miller-rabin-test 7 10)) (newline)
(display (miller-rabin-test 0 10)) (newline)
(display (miller-rabin-test 1 10)) (newline)
(display (miller-rabin-test 4 10)) (newline)
(display (miller-rabin-test 6 10)) (newline)
(display (miller-rabin-test 8 10)) (newline)
(display (miller-rabin-test 9 10)) (newline)
(display (miller-rabin-test 561 10)) (newline)
(display (miller-rabin-test 1105 10)) (newline)
(display (miller-rabin-test 1729 10)) (newline)
(display (miller-rabin-test 2465 10)) (newline)
(display (miller-rabin-test 2821 10)) (newline)
(display (miller-rabin-test 6601 10)) (newline)
```

This exercise is based on [sicp-ex-1.27](#).

```
;; ex 1.28

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

```

(define (miller-rabin-expmod base exp m)
  (define (squaremod-with-check x)
    (define (check-nontrivial-sqr1 x square)
      (if (and (= square 1)
                (not (= x 1))
                (not (= x (- m 1))))
          0
          square))
    (check-nontrivial-sqr1 x (remainder (square x) m)))
  (cond ((= exp 0) 1)
        ((even? exp) (squaremod-with-check
                      (miller-rabin-expmod base (/ exp 2) m)))
        (else
         (remainder (* base (miller-rabin-expmod base (- exp 1) m))
                   m)))))

(define (miller-rabin-test n)
  (define (try-it a)
    (define (check-it x)
      (and (not (= x 0)) (= x 1)))
    (check-it (miller-rabin-expmod a (- n 1) n)))
  (try-it (+ 1 (random (- n 1)))))

(define (fast-prime? n times)
  (cond ((= times 0) true)
        ((miller-rabin-test n) (fast-prime? n (- times 1)))
        (else false)))

(define (prime? n)
; Perform the test how many times?
; Use 100 as an arbitrary value.
(fast-prime? n 100))

(define (report-prime n expected)
  (define (report-result n result expected)
    (newline)
    (display n)
    (display ": ")
    (display result)
    (display ": ")
    (display (if (eq? result expected) "OK" "FOOLED")))
  (report-result n (prime? n) expected))

(report-prime 2 true)
(report-prime 7 true)
(report-prime 13 true)
(report-prime 15 false)
(report-prime 37 true)
(report-prime 39 false)

(report-prime 561 false) ; Carmichael number
(report-prime 1105 false) ; Carmichael number
(report-prime 1729 false) ; Carmichael number
(report-prime 2465 false) ; Carmichael number
(report-prime 2821 false) ; Carmichael number
(report-prime 6601 false) ; Carmichael number

```

Another solution that avoids nested functions. Similar to [http://wiki.drewhess.com/wiki/SICP\\_exercise\\_1.28](http://wiki.drewhess.com/wiki/SICP_exercise_1.28). However, this version does not intermingle the non-trivial-sqrt property check with returning the correct result in expmod. Instead, the non-trivial-sqrt property check is a dedicated function which returns true or false, as it should be. Only works for n>0. Uses the 'let' special form not introduced at this point of the book.

```

(define (miller-rabin n)
  (miller-rabin-test (- n 1) n))

(define (miller-rabin-test a n)
  (cond ((= a 0) true)
        ; expmod is congruent to 1 modulo n
        ((= (expmod a (- n 1) n) 1) (miller-rabin-test (- a 1) n))
        (else false)))

(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (let ((x (expmod base (/ exp 2) m)))
           (if (non-trivial-sqrt? x m) 0 (remainder (square x) m))))
        (else
         (remainder (* base (expmod base (- exp 1) m))
                   m)))))

(define (non-trivial-sqrt? n m)
  (cond ((= n 1) false)

```

```

( (= n (- m 1)) false)
; book reads: whose square is equal to 1 modulo n
; however, what was meant is square is congruent 1 modulo n
(else (= (remainder (square n) m) 1)))

```

Another **solution** in GNU Guile:

```

;;;; Under Creative Commons Attribution-ShareAlike 4.0
;;;; International. See
;;;; <https://creativecommons.org/licenses/by-sa/4.0/>.

(define-module (net ricketyspace sicp one twentyeight)
  #:use-module (srfi srfi-1)
  #:export (miller-rabin-test prime? run-tests))

(define (sqmod x m)
  "Return  $x^2 \bmod m$  if  $x^2 \bmod m$  is not equal to  $1 \bmod m$ 
and  $x \neq m - 1$  and  $x \neq 1$ ; 0 otherwise."
  (let ((square (* x x)))
    (cond ((and (= (remainder square m) 1) ; 1 mod m = 1
                (not (= x (1- m)))
                (not (= x 1)))
            0)
           (else square)))))

(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (sqmod (expmod base (/ exp 2) m) m)
                    m))
        (else
         (remainder (* base (expmod base (- exp 1) m))
                    m)))))

(define (miller-rabin-test n)
  (define (pass? a)
    (= (expmod a (1- n) n) 1))
  (fold (lambda (a p) (and (pass? a) p)) #t (iota (1- n) 1)))

(define (prime? n)
  (if (miller-rabin-test n) #t #f))

;; Tests

(define (carmichael-numbers-pass?)
  "Return #t if the sample carmichael numbers are detected as non-prime."
  (let ((numbers '(561 1105 1729 2465 2821 6601)))
    (cons "carmichael-numbers-pass?"
          (fold (lambda (n p) (and (not (prime? n)) p)) #t numbers)))))

(define (prime-numbers-pass?)
  "Return #t if the sample prime numbers are detected as prime"
  (let ((numbers '(311 641 829 599 809 127 419 13 431 883)))
    (cons "prime-numbers-pass?"
          (fold (lambda (n p) (and (prime? n) p)) #t numbers)))))

(define (even-numbers-pass?)
  "Return #t if the sample even numbers are detected as non-prime"
  (let ((numbers '(302 640 828 594 804 128 414 12 436 888)))
    (cons "prime-numbers-pass?"
          (fold (lambda (n p) (and (not (prime? n)) p)) #t numbers)))))

(define (run-tests)
  (map (lambda (test) (test)) (list carmichael-numbers-pass?
                                       prime-numbers-pass?
                                       even-numbers-pass?)))

;; Guile REPL
;;
;; scheme@(guile-user)> ,use (net ricketyspace sicp one twentyeight)
;; scheme@(guile-user)> (run-tests)
;; $18 = (("carmichael-numbers-pass?" . #t)
;;        ("prime-numbers-pass?" . #t)
;;        ("prime-numbers-pass?" . #t))

```

tiendo1011

When  $n = 9$ , the test doesn't return 1 for any  $a \leq n - 1$ . Which makes me wonder the validity of the following claim:

It is also possible to prove that if  $n$  is an odd number that is not prime, then, for at least half the numbers  $a < n$ , computing  $a^{n-1}$  in this way will reveal a nontrivial square root of 1 modulo  $n$ .

So I dig deeper and find this answer on StackOverflow:  
<https://stackoverflow.com/a/59834347/4632735> Here is my attempt trying to convert the answer to scheme code

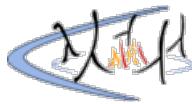
```
(define (display-all . vs)
  (for-each display vs))

(define (find-e-k n)
  (define (find-e-k-iter possible-k possible-e)
    (if (= (remainder possible-k 2) 0)
        (find-e-k-iter (/ possible-k 2) (+ possible-e 1))
        (values possible-e possible-k)))
  (find-e-k-iter (- n 1) 0))

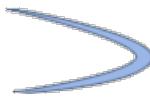
; first-witness-case-test: (a ^ k) mod n # 1
(define (first-witness-case-test a k n)
  (not (= (expmod a k n) 1)))

; second-witness-case-test: all a ^ ((2 ^ i) * k) (with i = {0..e-1}) mod n # (n - 1)
(define (second-witness-case-test a e k n)
  (define (second-witness-case-test-iter a i k n)
    (cond ((= i -1) true)
          (else (let ()
                  (define witness (not (= (expmod a (* (fast-expt 2 i) k) n) (- n 1))))
                  (if witness
                      (second-witness-case-test-iter a (- i 1) k n)
                      false))))))
  (second-witness-case-test-iter a (- e 1) k n))

(define (miller-rabin-test n)
  (define (try-it a e k)
    (if (and (first-witness-case-test a k n) (second-witness-case-test a e k n))
        (display-all "is not prime, with a = " a "\n")
        (if (< a (- n 1))
            (try-it (+ a 1) e k)
            (display "is prime\n"))))
  (cond ((< n 2) (display "not prime"))
        ((= (remainder n 2) 0) (display "not prime\n"))
        (else (let ()
                  (define-values (e k) (find-e-k n))
                  (try-it 1 e k)))))
```



# sicp-ex-1.29



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (1.28) | sicp-solutions | Next exercise (1.30) >>

Maggyero

## QUESTION

Simpson's Rule is a more accurate method of numerical integration than the method illustrated above. Using Simpson's Rule, the integral of a function  $f$  between  $a$  and  $b$  is approximated as

$$h/3 (y_0 + 4 y_1 + 2 y_2 + 4 y_3 + 2 y_4 + \dots + 2 y_{n-2} + 4 y_{n-1} + y_n)$$

where  $h = (b - a)/n$ , for some even integer  $n$ , and  $y_k = f(a + kh)$ . (Increasing  $n$  increases the accuracy of the approximation.) Define a procedure that takes as arguments  $f$ ,  $a$ ,  $b$ , and  $n$  and returns the value of the integral, computed using Simpson's Rule. Use your procedure to integrate cube between 0 and 1 (with  $n = 100$  and  $n = 1000$ ), and compare the results to those of the integral procedure shown above.

## ANSWER

```
(import (scheme small))

(define (simpson-integral f a b n)
  (define (sum term a next b)
    (if (> a b)
        0
        (+ (term a) (sum term (next a) next b))))
  (define (term x)
    (+ (f x) (* 4 (f (+ x h))) (f (+ x (* 2 h))))))
  (define (next x)
    (+ x (* 2 h)))
  (define h (/ (- b a) n))
  (* (/ h 3) (sum term a next (- b (* 2 h)))))

(define (cube x)
  (* x x x))

(display (simpson-integral cube 0 1 100)) (newline)
(display (simpson-integral cube 0 1 1000)) (newline)
```

The integral of cube between 0 and 1 computed using the composite Simpson's rule is exact because the error of that method is proportional to the fourth derivative of the function to integrate, so for cube the error is zero. The integral of cube between 0 and 1 computed using the composite midpoint rule is not exact because the error of that method is proportional to the second derivative of the function to integrate, so for cube the error is nonzero.

```
(define (round-to-next-even x)
  (+ x (remainder x 2)))
```

```
(define (simpson f a b n)
  (define fixed-n (round-to-next-even n))
  (define h (/ (- b a) fixed-n))
  (define (simpson-term k)
    (define y (f (+ a (* k h))))
    (if (or (= k 0) (= k fixed-n))
        (* 1 y)
        (if (even? k)
            (* 2 y)
            (* 4 y))))
    (* (/ h 3) (sum simpson-term 0 inc fixed-n)))
```

This is a similar solution, complete with testing code. There is no rounding of  $n$  to the next even number, because the exercise assumes  $n$  to be even.

```
;; ex 1.29

(define (cube x) (* x x x))
```

```

(define (inc n) (+ n 1))

(define (sum term a next b)
  (if (> a b)
    0
    (+ (term a)
        (sum term (next a) next b)))))

(define (simpson-integral f a b n)
  (define h (/ (- b a) n))
  (define (yk k) (f (+ a (* h k))))
  (define (simpson-term k)
    (* (cond ((or (= k 0) (= k n)) 1)
              ((odd? k) 4)
              (else 2))
            (yk k)))
    (* (/ h 3) (sum simpson-term 0 inc n)))

;; Testing
(simpson-integral cube 0 1 100)
(simpson-integral cube 0 1 1000)

```

There is a third way which approaches the solution by breaking the problem into four parts: ( $f(y_0)$ , ( $f(y_n)$ ) and two sums, one over even  $k$  and another over odd  $k$ .

```

(define (another-simpson-integral f a b n)
  (define h (/ (- b a) n))
  (define (add-2h x) (+ x (* 2 h)))
  (* (/ h 3.0) (+ (f a)
                    (* 4.0 (sum f (+ a h) add-2h b))
                    (* 2.0 (sum f (add-2h a) add-2h (- b h)))
                    (f b))))
```

Here's a version that sums over pairs of terms ( $2 y_k + 4 y_{k+1}$ ). No conditionals or special cases are needed anywhere, but there's an extra term [ $f(b) - f(a)$ ] to be added to the final count.

```

(define (simpson f a b n)
  (define h (/ (- b a) n))

  (define (y k)
    (f (+ a (* k h)))))

  (define (ypair k)
    (+ (* 2 (y k))
       (* 4 (y (+ k 1)))))

  (define (add-2 k)
    (+ k 2))

  (* (/ h 3) (+ (sum ypair 0 add-2 (- n 1))
                 (- (f b) (f a)))))
```

```

(define (sim-integral f a b n)
  (define h (/ (- a b) n))
  (define (y k) (f (+ a (* k h))))
  (define (coeff k) (if (is-even? k) 2 4))
  (define (part-term k) (* (coeff k) (y k)))
  (define part-value (sum part-term 1 inc (- n 1)))
  (* (/ h 3) (+ (y 0) (y n) part-value)))
```

Another **solution** in GNU Guile:

```

;;;; Under Creative Commons Attribution-ShareAlike 4.0
;;;; International. See
;;;; <https://creativecommons.org/licenses/by-sa/4.0/>.

(define-module (net ricketyspace sicp one twentynine)
  #:export (simpson))

(define (sum term a next b)
  (if (> a b)
    0
    (+ (term a)
        (sum term (next a) next b)))))

(define (simpson f a b n)
  (let* ((h (/ (- b a) (* 1.0 n)))
         (y (lambda (k) (+ a (* k h)))))
         (ce (lambda (k) :coefficient
                  (cond ((or (= k 0) (= k n)) 1)
                        ((even? k) 2)
                        (else 4))
                  (yk k)))
         (part-term k) (* (ce k) (y k)))
         (part-value (sum part-term 1 inc (- n 1)))
         (y0 (y 0))
         (yn (y n)))
    (+ y0 yn (* (/ h 3) part-value))))
```

```

        (else 4))))
  (term (lambda (k)
    (* (ce k) (f (y k)))))
  (next (lambda (k) (1+ k))))
(* (/ h 3.0)
  (sum term 0 next n)))

;; Guile REPL
;; scheme@(guile-user)> ,use (net ricketyspace sicp one twentynine)
;; scheme@(guile-user)> (define (cube x) (* x x x))
;; scheme@(guile-user)> (simpson cube 0 1 100)
;; $23 = 0.2499999999999992
;; scheme@(guile-user)> (simpson cube 0 1 1000)
;; $24 = 0.2500000000000003

```

While testing the integral example presented before the ex 1.29 I had some problems for small dx; Aborting!: maximum recursion depth exceeded]. So I tried a iterative version :

```

(define (sum-iter ans foo a next b)
  (if (> a b)
    ans
    (sum-iter (+ ans (foo a)) foo (next a) next b)))

```

Using sum-iter in Simpson Integral definition:

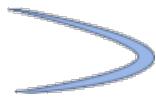
```

(define (simpson-integral foo a b n)
  (define h (/ (- b a) n))
  (define (y k) (foo (+ a (* k h))))
  (define (intersimp k) (* (cond ((or (= k 0) (= k n)) 1)
    ((even? k) 2)
    (else 4)))
    (y k)))
  (define (inc a) (+ a 1))
  (* (sum-iter 0 intersimp 0 inc n) (/ h 3.0)))
)

```



# sicp-ex-1.30



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

---

<< Previous exercise (1.29) | sicp-solutions | Next exercise (1.31) >>

---

```
(define (itersum term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (+ result (term a)))))
  (iter a 0))
```

To Test

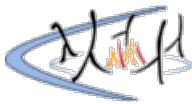
```
(define (pi-sum a b)
  (define (pi-term x)
    (/ 1.0 (* x (+ x 2))))
  (define (pi-next x)
    (+ x 4))
  (itersum pi-term a pi-next b))
```

```
(* 8 (pi-sum 1 1000))
```

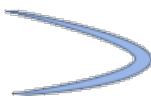
should = 3.139592655589783

---

Last modified : 2017-11-09 15:00:18  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-1.31



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (1.30) | sicp-solutions | Next exercise (1.32) >>

a.

```
(define (product term a next b)
  (if (> a b) 1
      (* (term a) (product term (next a) next b))))
```

factorial function, in terms of product function above, can be written as below.

```
(define (identity x) x)
(define (next x) (+ x 1))
(define (factorial n)
  (product identity 1 next n))
```

approximations to pi using john wallis' formula, can be found by defining a new term to be used by product function as below

```
(define (pi-term n)
  (if (even? n)
      (/ (+ n 2) (+ n 1))
      (/ (+ n 1) (+ n 2))))
```

And it can be used as follows.

```
(* (product pi-term 1 next 6) 4) ;;;= 3.3436734693877552
(* (product pi-term 1 next 100) 4) ;;;= 3.1570301764551676
```

b.

```
(define (product term a next b)
  (define (iter a res)
    (if (> a b) res
        (iter (next a) (* (term a) res))))
  (iter a 1))
```

Another approach would be to recognize the series as  $4 \cdot 4 \cdot 6 \cdot 6 \cdots / 3 \cdot 3 \cdot 5 \cdot 5 \cdots$ . The leading factor of 2 in the numerator must be dealt with.

```
(define (pi n)
  (define (term x) (* x x))
  (define (next x) (+ x 2))
  ; since we are increasing the numbers by two on every iteration
  (define limit (* n 2))
  ; upper term: - 2 always goes first, start building product from 4
  ;             - as 2 numbers are skipped, the limit must respect that, too
  ;             - since we are squaring one time too often at the end,
  ;               we have to divide that back out of the result
  ; lower term: start with 3, which is 1 more than the upper term
  ;             -> so increase limit by 1
  ;
  (* 4 (/ (/ (* 2 (product term 4 next (+ limit 2)))
    (+ limit 2))
    (product term 3 next (+ limit 1)))))
```

Yielding:

```
(pi 6) ;;;= 3.255721745
(pi 100) ;;;= 3.149378473
```

n defines steps in terms of pairs of factors. So where the above solution does 6 steps when n=6, this solution does 12 steps when n=6:

```
(pi 3) ;:= 3,343673469
(pi 50) ;:= 3,157030176
```

A third way of defining the pi-term function, using integer arithmetic instead of the even? predicate:

```
(define (pi n)
  (define (pi-term n)
    (/ (* 2 (ceiling (/ (+ 1 n) 2)))
       (+ 1 (* 2 (ceiling (/ n 2)))))))
  (product pi-term 1 inc n))
```

A fourth way of doing it is to recognize  $\pi = 4 * (2 * (1/3) * 4 * (1/5) ...) * ((1/3) * 4 * (1/5) ...)$

Arithmetic progression formula was used to generate the nth number of the sequence

namely  $a_k = a_1 + (k-1)d$

```
(define (pi-prod n)
  (define (inv k)
    (if (even? k)
        k
        (/ 1 k)))
  (* 4.0 (* (product 2 (+ 2 (- n 1)) inc inv)
             (product 3 (+ 3 (- n 1)) inc inv))))
```

Another way is to recognise that the formula for  $\pi$  is composed of two separate products:

$\pi = 4 * ((2/3 * 4/5 * 6/7 \dots * n/(n+1)) * (4/3 * 6/5 * \dots * n/(n-1)))$

This gives:

```
(define (approx-pi n)
  (* 4
     (product
      (lambda (n) (/ n (+ 1 n)))
      2
      (lambda (n) (+ n 2))
      n)
     (product
      (lambda (n) (/ n (- n 1)))
      4
      (lambda (n) (+ n 2))
      n)))
```

master

Here's how I visualize the problem: If you recognize that the denominator follows the pattern 3,3,5,5,7,7 and that for any given pair of n's, the numerators will be n-1 and n+1, then there is no need to take into account the "stray" 2 at the start of the numerator sequence.

```
(define (product-iter f a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (* (f a) result))))
  (iter a 1))

(define (pi-term n)
  (* (/ (- n 1.0) n) (/ (+ n 1.0) n)))

(define (next-pi n)
  (+ n 2.0))

(define (pi accuracy)
  (* 4 (product-iter pi-term 3.0 next-pi accuracy)))
```

My method is the same as the above. A clarification though, this method works by splitting the formula into:

$(2 \cdot 4/3 \cdot 3) \cdot (4 \cdot 6/5 \cdot 5) \cdot (6 \cdot 8/7 \cdot 7)$

We can then see the general expression for each pair of terms:

$((n-1) \cdot (n+1))/n^2 \rightarrow (n^2 - 1)/n^2 \rightarrow n^2/n^2 - 1/n^2 \rightarrow 1 - 1/n^2$

This translates into code as:

```
(define (pi-term a)
  (- 1 (/ 1 (square a)))))

(define (pi-next a)
  (+ a 2))

(define (pi accuracy)
  (* 4.0
     (product-iter pi-term 3 pi-next (+ (* accuracy 2) 1))))
```

The method I used for the Wallis Product recognizes the pairs as following:

$(2/3) \rightarrow 2.5$

$(4/3) \rightarrow 3.5$

$(4/5) \rightarrow 4.5$

$(6/5) \rightarrow 5.5$

...

We first of all define our sum (named Pi as in the mathematical notation):

```
(define (pi term next n upper_bound)
  (if (> n upper_bound)
      1
      (* (term n)
         (pi term next (next n) upper_bound)
         )
      )
  )
```

Then we define the Wallis Product:

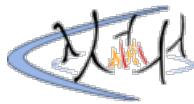
```
(define (wallis a b pi)
  (define (wallis-term x)
    (if (= (remainder (inexact->exact (floor x)) 2) 0)
        (/ (floor x) (ceiling x))
        (/ (ceiling x) (floor x)))
    )
  )

  (define (wallis-next x)
    (+ x 1)
  )

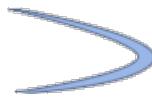
  (* 4 (pi wallis-term wallis-next a b))
)
```

And then we can test it:

```
(display (wallis 2.5 1000 pi))
```



# sicp-ex-1.32



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

---

<< Previous exercise (1.31) | sicp-solutions | Next exercise (1.33) >>

---

## a. Accumulate procedure

Recursive process:

```
; right fold
(define (accumulate combiner null-value term a next b)
  (if (> a b) null-value
      (combiner (term a) (accumulate combiner null-value term (next a) next b))))
```

Sum and product as simple calls to accumulate:

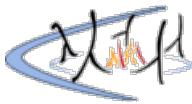
```
(define (sum term a next b) (accumulate + 0 term a next b))
(define (product term a next b) (accumulate * 1 term a next b))
```

## b. Iterative process:

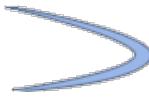
```
; left fold
(define (accumulate combiner null-value term a next b)
  (define (iter a res)
    (if (> a b) res
        (iter (next a) (combiner res (term a))))))
  (iter a null-value))
```

---

Last modified : 2017-11-09 15:00:54  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-1.33



[\[Top Page\]](#) [\[Recent Changes\]](#) [\[All Pages\]](#) [\[Settings\]](#) [\[Categories\]](#) [\[Wiki Howto\]](#)  
[\[Edit\]](#) [\[Edit History\]](#)  
 Search:

**<< Previous exercise (1.32) | sicp-solutions | Next exercise (1.34) >>**

a. The filtered\_accumulate procedure is not optimized at all.. We'll use the *prime?* procedure defined earlier in the book with a small fix to make (*prime?* 1) -> #f.

```
(define (smallest-div n)
  (define (divides? a b)
    (= 0 (remainder b a)))
  (define (find-div n test)
    (cond ((> (sq test) n) n) ((divides? test n) test)
          (else (find-div n (+ test 1)))))
  (find-div n 2))

(define (prime? n)
  (if (= n 1) false (= n (smallest-div n))))
```

Filtered accumulate procedure:

```
(define (filtered-accumulate combiner null-value term a next b filter)
  (if (> a b) null-value
      (if (filter a)
          (combiner (term a) (filtered-accumulate combiner null-value term (next a) next b
filter))
          (combiner null-value (filtered-accumulate combiner null-value term (next a) next b
filter)))))
```

A slightly dryer recursive version:

```
(define (filtered-accumulator filter combiner null-value term a next b)
  (if (> a b) null-value
      (combiner (if (filter a) (term a)
                   null-value)
                (filtered-accumulator filter combiner null-value term (next a) next b))))
```

And an iterative version:

```
(define (filtered-accumulate combiner null-value term a next b filter)
  (define (iter a result)
    (cond ((> a b) result)
          ((filter a) (iter (next a) (combiner result (term a))))
          (else (iter (next a) result))))
  (iter a null-value))
```

a. Sum of squares of prime numbers procedure:

```
(define (sum-of-prime-squares a b) (filtered-accumulate + 0 sq a inc b prime?))
```

Test example: (sum-of-prime-squares 1 5)

ans. 38

b. Product of all positive integers les than n that are relatively prime to n

```
(define (gcd m n)
  (cond ((< m n) (gcd n m))
        ((= n 0) m)
        (else (gcd n (remainder m n)))))

(define (relative-prime? m n)
  (= (gcd m n) 1))

(define (product-of-relative-primes n)
  (define (filter x)
    (relative-prime? x n))
  (filtered-accumulate * 1 identity 1 inc n filter))
```

Example: (product-of-relative-primes 10) 189

poly

the solutions above:

```
(define (filtered-accumulate combiner null-value term a next b filter)
  (if (> a b) null-value
      (if (filter a)
          (combiner (term a) (filtered-accumulate combiner null-value term (next a) next b
filter))
          (combiner null-value (filtered-accumulate combiner null-value term (next a) next
b filter)))))
```

There is no need to combine null-value with the later accumulating actually.

```
(define (filtered-accumulate filter combiner null-value term a next b)
  (if (> a b)
      null-value
      (if (filter a)
          (combiner (term a)
                    (filtered-accumulate filter combiner null-value term (next a) next
b)))
          (filtered-accumulate filter combiner null-value term (next a) next b)))))

; so the accumulate can be:
(define (accumulate combiner null-value term a next b)
  (filtered-accumulate (lambda (x) true)
                      combiner null-value term a next b))
```

BTW: I can't stand the indent of the above guy's codes.

cgb

a way to avoid code repetition both in the recursive and in the iterative version is to add a conditional in the first term of combiner. This way there is no need to call two versions of filtered-accumulate. The recursive version will be:

```
(define (filtered-accumulate combiner null-value term a next b filter)
  (if (> a b)
      null-value
      (combiner (if (filter a)
                    (term a)
                    null-value)
                (filtered-accumulate combiner null-value term (next a) next b filter))))
```

And the iterative version:

```
(define (filtered-accumulate combiner null-value term a next b filter)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (combiner result
                                  (if (filter a)
                                      (term a)
                                      null-value)))))

  (iter a null-value))
```

ctz

An alternative way making use of the accumulate procedure before:

```
(define (filtered-accumulate filter comb null-val term a next b)
  (define (filtered-term k)
    (if (filter k) (term k) null-val))
  (accumulate comb null-val filtered-term a next b))
```

@poly: I don't see the point of defining accumulate in terms of filtered-accumulate.

# sicp-ex-1.34

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

<< Previous exercise (1.33) | sicp-solutions | Next exercise (1.35) >>

---

First invocation of `f` will attempt to apply its argument (which is `f`) to 2. This second invocation will attempt to apply its argument (which is 2) to 2, resulting in error.

```
(f f)
(f 2)
(2 2)
; Error
; MIT Scheme reports: The object 2 is not applicable.
```

Note that both substitution models, *applicative-order* evaluation and *normal-order* evaluation, will lead to the same expansion.

---

Last modified : 2017-11-09 15:01:51  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-1.35

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

<< Previous exercise (1.34) | sicp-solutions | Next exercise (1.36) >>

---

First, we prove that the transformation  $x \rightarrow 1 + 1/x$  yields phi:

```
x = 1 + 1/x
x^2 = x + 1
```

This second equation is the definition of phi from 1.2.2. We can also derive the explicit formula for phi from this equation by recasting as a quadratic and solving with the quadratic formula (discarding the negative root):

```
x^2 - x - 1 = 0
x = ( 1 + sqrt(5) ) / 2
```

The code is thus simple:

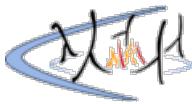
```
(fixed-point (lambda (x) (+ 1 (/ 1 x))) 1.0)
```

Sample run on MzScheme with routine to print x and f(x) in (fixed-point):

```
x = 1.0 and f(x) = 2.0
x = 2.0 and f(x) = 1.5
x = 1.5 and f(x) = 1.6666666666666665
x = 1.6666666666666665 and f(x) = 1.6
x = 1.6 and f(x) = 1.625
x = 1.625 and f(x) = 1.6153846153846154
x = 1.6153846153846154 and f(x) = 1.619047619047619
x = 1.619047619047619 and f(x) = 1.6176470588235294
x = 1.6176470588235294 and f(x) = 1.61818181818182
x = 1.61818181818182 and f(x) = 1.6179775280898876
x = 1.6179775280898876 and f(x) = 1.6180555555555556
x = 1.6180555555555556 and f(x) = 1.6180257510729614
x = 1.6180257510729614 and f(x) = 1.6180371352785146
x = 1.6180371352785146 and f(x) = 1.6180327868852458
x = 1.6180327868852458 and f(x) = 1.618034447821682
x = 1.618034447821682 and f(x) = 1.618033813400125
x = 1.618033813400125 and f(x) = 1.6180340557275543
x = 1.6180340557275543 and f(x) = 1.6180339631667064
x = 1.6180339631667064 and f(x) = 1.6180339985218035
x = 1.6180339985218035 and f(x) = 1.618033985017358
x = 1.618033985017358 and f(x) = 1.6180339901755971
```

Another solution using average damping:

```
(fixed-point (lambda (x) (average x (+ 1 (/ 1 x)))) 1.0)
```



# sicp-ex-1.36



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

---

<< Previous exercise (1.35) | sicp-solutions | Next exercise (1.37) >>

---

For the modification to print each value:

```
(define tolerance 0.000001)

(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (display guess)
    (newline)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

Note that it would be more idiomatic to extract the small function:

```
(define (print-line value)
  (display value)
  (newline))
```

```
(define (x-to-the-x y)
  (fixed-point (lambda (x) (/ (log y) (log x)))
              10.0))
```

---

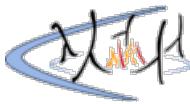
Using my version of Scheme (Petite Chez Scheme), this takes 33 iterations to converge, printing out the final answer on the 34th line.

If we make the suggested change to use the average function,

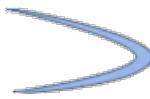
```
(define (x-to-the-x y)
  (fixed-point (lambda (x) (average x (/ (log y) (log x)))))
              10.0))
```

---

This converges in 10 iterations, printing the result on the 11th line.



# sicp-ex-1.37



[\[Top Page\]](#) [\[Recent Changes\]](#) [\[All Pages\]](#) [\[Settings\]](#) [\[Categories\]](#) [\[Wiki Howto\]](#)  
[\[Edit\]](#) [\[Edit History\]](#)  
 Search:

[\*\*<< Previous exercise \(1.36\) | sicp-solutions | Next exercise \(1.38\) >>\*\*](#)

a) An iterative solution is:

```
(define (cont-frac n d k)
  (define (loop result term)
    (if (= term 0)
        result
        (loop (/ (n term)
                  (+ (d term) result))
              (- term 1))))
  (loop 0 k))
```

It takes on the order of ten terms to be accurate within 0.0001.

b) Making the solution recursive:

```
(define (cont-frac n d k)
  (cond ((= k 0) 0)
        (else (/ (n k) (+ (d k) (cont-frac n d (- k 1)))))))
```

This requires the same number of terms to be accurate within 0.0001.

The recursive solution above is incorrect, it produces a continued fraction with the index values \*decreasing\* as you descend into each layer of denominators instead of increasing. i.e. :  $N_k / (D_k + (N_{k-1} / (D_{k-1} + \dots)))$  etc. The test does not expose this since the procedures passed to it as n and d always return 1.0 regardless of the (index) values passed to them.

A correct recursive version requires the recursive procedure to be defined inside the cont-frac procedure in order to use an ascending index value, e.g. :

```
(define (cont-frac n d k)
  (define (frac-rec i)
    (/ (n i)
       (+ (d i)
          (if (= i k)
              0
              (frac-rec (+ i 1)))))))
  (frac-rec 1))
```

Iterative version:

```
(define (cont-frac n d k)
  (define (cont-frac-iter i result)
    (if (= i 0)
        result
        (cont-frac-iter (- i 1)
                      (/ (n i) (+ (d i) result)))))
  (cont-frac-iter k 0.0))
```

This is incorrect. It needs to do the last division. Like This:

```
(define (cont-frac n d k)
  (define (cont-frac-iter i result)
    (if (= i 0)
        (/ (n i) (+ (d i) result))
        (cont-frac-iter (- i 1)
                      (/ (n i) (+ (d i) result)))))
  (cont-frac-iter k 0.0))
```

Another way to accomplish the same result, is by calling it like this the first time:

```
(cont-frac-iter k (/ (n k) (d k)))
```

instead of using 0.0. And have it return only `result` at the end.

---

Actually, correction to your correction. The "last division" is already taken care of with i=1. If you see the text, the definition uses i=1 to i=k. There is no i=0. So by the time i is decremented to 0, result already has the correct answer. You do not need to do any more divisions.

---

The recursive solution above is incorrect also. It calculate the finite continued fraction from 1 to k-1, instead of k. A minor fix is needed:

```
(define (cont-fact n d k)
  (define (recur i)
    (if (> i k)
        0
        (/ (n i) (+ (d i) (recur (add1 i)))))))
  (recur 1))
```

Iterative version is the same.

---

A recursive solution that doesn't require a helper function, but is somewhat awkward:

```
(define (cont-frac n d k)
  (if (= k 0)
      0
      (/ (n 1)
          (+ (d 1)
              (cont-frac
                (lambda (i) (n (+ i 1)))
                (lambda (i) (d (+ i 1)))
                (- k 1)))))))
```

---

None of the iterative solutions above work because there is no iterative solution. There is no way to accumulate a partial result and pass it forward because the ultimate result will always depend on all terms in a single unwound expression.

This is not apparent due to the test case's n term and d term being unchanging with i. If instead we use the (ex 1.38) e-euler test case we will calculate the incorrect value of 2.50373... rather than 2.71828... I am using:

(define (cont-frac n d k)

```
(define (frac-iter i result)
  (if (> i k)
      result
      (frac-iter (+ i 1) (/ (n i) (+ (d i) result)))))

  (frac-iter 1 0.0))
```

for the failed case yielding 2.50373... and

(define (cond-frac n d k)

```
(define (frac-rec i result)
  (if (> i k)
      0
      (/ (n i) (+ (d i) (frac-rec (+ i 1)))))

  (frac-rec 1))
```

for the passed case yielding 2.71828...

---

The comment above is wrong: to construct an iterative solution, you neek to work backwards, starting with (n k) and (d k) and working your way down with (n (- k 1)) and (d (- k 1)) etc. Here is a minor fix that makes the above solution work:

```
(define (cont-frac n d k)
  (define (frac-iter i result)
    (if (= i k)
        result
        (frac-iter (+ i 1) (/ (n (- k i)) (+ (d (- k i)) result)))))

  (frac-iter 0 0.0))
```

Note that because you start counting at 0, you need to stop the counter at k (not k + 1).

---

Actually, I think that the recursive version of the first comment and the iterative version of the second comment is already correct. Perhaps the discussion around this exercise is carried to too great a length.

Let's clear it up. The problem asks for layers of fractions, which can be achieved either by a recursive process or by an iterative process. The number of steps would be k. Now, the tricky part is sequence of the count from 1 to k. For the recursive process, the fraction is built, in a way, from the outside to the inside. So the count should go from 1 to k. For the iterative process, however, the fraction accumulates from the inside to the outside, and the count should consequently go from k to 1.

Below is both versions:

```
(define (cont-frac n d k)
  (define (rec x)
    (if (> x k)
        0
        (/ (n x) (+ (d x) (rec (+ x 1))))))
  (rec 1))
```

```
(define (cont-frac n d k)
  (define (iter x result)
    (if (= x 0)
        result
        (iter (- x 1) (/ (n x) (+ (d x) result)))))

  (iter k 0)))
```

This sequence from 1 to k or from k to 1 is not important in the context of this exercise, but for situations where the value of n or d depend on the input, the sequence does matter.

---

Yes. If the value of n or d depend on the input, recursive version should look forward while the iterative version should look backward.

# sicp-ex-1.38

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (1.37) | sicp-solutions | Next exercise (1.39) >>

```
(define (e-euler k)
  (+ 2.0 (cont-frac (lambda (i) 1)
                      (lambda (i)
                        (if (= (remainder i 3) 2)
                            (/ (+ i 1) 1.5)
                            1))
                      k)))
```

:Alternative solution.

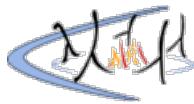
```
; There is a repeating pattern with a cycle of 3 in the sequence of values of d.
; The first value is 1, the second a power of 2 and the third is again a 1
; To find out the relative place of a value in a cycle take the index of the
; value in the sequence modulus 3.
; Further note the second value within a cycle goes up from 2 to 2+2 to 2+2+2 ...
```

```
(define (d i) ;
  (define (d-iter value times-two j)
    (let ((j-mod-3 (modulo j 3)))
      (if (> j i)
          value
          (d-iter
            (cond ((= j-mod-3 1) 1)
                  ((= j-mod-3 2) times-two)
                  ((= j-mod-3 0) 1)))
            (if (= j-mod-3 2)
                (+ times-two 2)
                times-two)
            (+ j 1))))
        (d-iter 0 2 1)))

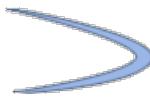
(define (euler-e) ;multiplication by 1.0 forces fraction to real
  (* 1.0 (+ (finite-cont-frac n d 100) 2)))

(define (enum f n) ;displays the function values upto and including n
  (define (enum-iter list-of-values i)
    (if (= i n)
        (display (reverse list-of-values))
        (enum-iter (cons (f i) list-of-values) (+ i 1))))
  (enum-iter '() 1))
```

Last modified : 2018-01-06 22:08:49  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-1.39



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (1.38) | sicp-solutions | Next exercise (1.40) >>

```
(define (tan-cf x k)
  (cont-frac (lambda (i)
                (if (= i 1) x (- (* x x))))
              (lambda (i)
                (- (* i 2) 1))
              k))
```

Another version calculating the square only once using let.

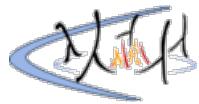
```
(define (tan-cf x k)
  (let ((a (- (* x x))))
    (cont-frac (lambda (i) (if (= i 1) x a))
               (lambda (i) (- (* i 2) 1))
               k)))
```

Dividing once by  $-x$  outside the call to cont-frac obviates the need for the *if* statement and allows a static return value of  $-(x^2)$  for the  $n$  argument to cont-frac.

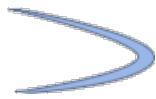
```
(define (tan-cf x k)
  (let ((X (- (* x x))))
    (- (/ (cont-frac (lambda (i) X)
                      (lambda (i) (- (* i 2) 1))
                      k)
          x)))
  ))
```

Here's yet another version, not defined in terms of cont-frac.

```
(define (tan-cf x k)
  (define (tan-cf-rec i)
    (let ((di (+ i (- i 1)))
          (x^2 (* x x)))
      (if (= i k)
          di
          (- di (/ x^2 (tan-cf-rec (+ i 1)))))))
  (/ x (tan-cf-rec 1)))
```



# sicp-ex-1.40



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

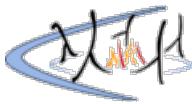
---

<< Previous exercise (1.39) | sicp-solutions | Next exercise (1.41) >>

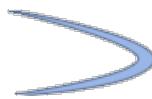
---

```
(define (cubic a b c)
  (lambda (x)
    (+ (cube x)
       (* a (square x))
       (* b x)
       c)))
```

Last modified : 2017-11-09 15:11:40  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-1.41



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (1.40) | sicp-solutions | Next exercise (1.42) >>

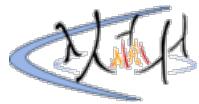
```
(define (double f)
  (lambda (x) (f (f x))))
((double (double double)) inc) 5)
```

I've got this explanation to fully understand what is happening behind the scene

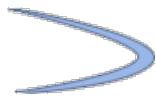
```
; explicacion:
double1(f) = f (f)
double2 double1 = double1 (double1)
double3 double2 = double2 (double2) = double1 (double1) (double1 (double1))
double1(inc) = inc(inc)
double1 (double1) (double1 double1(inc))
double1 (double1) (double1 (inc(inc)))
double1 (f (f)) (inc(inc(inc(inc))))
f(f(f(f))) (inc(inc(inc(inc))))
16 times inc ; en español: 16 veces inc
=> 16 times inc(5) = 21
```

if we have n doubles, there will be  $2^{\{2^{(n-1)}}\}$  times inc, not  $2^n$  times inc.

Last modified : 2022-11-10 01:39:01  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-1.42



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

---

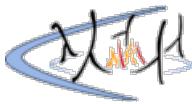
<< Previous exercise (1.41) | sicp-solutions | Next exercise (1.43) >>

---

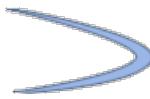
```
(define (compose f g)
  (lambda (x) (f (g x))))
```

---

Last modified : 2017-11-09 15:13:09  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-1.43



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (1.42) | sicp-solutions | Next exercise (1.44) >>

Define some primitives:

```
(define (square x) (* x x))
(define (compose f g) (lambda (x) (f (g x))))
```

Define the procedure:

```
(define (repeat f n)
  (if (< n 1)
      (lambda (x) x)
      (compose f (repeat f (- n 1)))))
```

Test with:

```
((repeat square 2) 5)
```

Output:

```
625
```

Another solution using the linear iterative way.

```
(define (repeat f n)
  (define (iter n result)
    (if (< n 1)
        result
        (iter (- n 1) (compose f result))))
  (iter n identity))
```

Note: This is not linearly iterative as described in the book as a chain of *deferred operations* is still being built.

The above answer does not follow the book's instructions. The book instructs "Write a procedure that takes as inputs a procedure that computes f and a positive integer n and returns the procedure that computes the nth repeated application of f." A correct answer is as follows:

```
(define (repeated f n)
  (lambda (x) (cond ((= n 0) x)
                      (else
                        ((compose (repeated f (- n 1)) f) x)))))
```

I think the following solution is more elegant. When we only need to apply the function once, we can just return the function (I'm not doing error-checking here to see if n is smaller than 1).

```
(define (repeated f n)
  (if (= n 1)
      f
      (compose f (repeated f (- n 1)))))
```

An extremely succinct solution uses the accumulate procedure defined in 1.32:

```
(define (repeated f n)
  (accumulate compose identity (lambda (i) f) 1 inc n))
```

An solution with O(log n) complexity using compose:

```
(define (repeated f n)
  (cond ((= n 0) identity)
```

```
((even? n) (repeated (compose f f) (/ n 2)))
(else (compose f (repeated f (- n 1)))))
```

A logarithmic iterative solution.

```
(define (lcompose f g)
  (lambda (x) (g (f x))))

(define (identity x) x)

(define (repeated f n)
  (define (iter g h m)
    (cond ((= m 0) g)
          ((odd? m) (iter (lcompose g h) h (- m 1)))
          (else (iter g (lcompose h h) (/ m 2)))))

  (iter identity f n)))
```

I personally would rather like to avoid using compose (as the only actual use I see in it would be in the logarithmic solution given above).

```
(define (repeated f n)
  (define (repeated-step counter input)
    (if (> counter n)
        input
        (f (repeated-step (+ counter 1) input)))
    )
  (lambda (x) (repeated-step 1 x))
)
```

What do you guys think about this way of doing it? I believe it generates a linear iterative process:

```
(define (repeated f n)
  (lambda (x)
    (define (iter n result)
      (cond ((= n 0) result)
            ((odd? n) (iter (- n 1) (f result)))
            (else
              (iter (- n 2) ((compose f f) result)))))

    (iter n x))))
```

Is this another correct way of solving? I get the right answer when testing the given example from the book: ((repeated square 2) 5) = 625. I know I'm missing something. New to Scheme and this course, so feedback/criticisms welcomed. Could someone provide a good explanation of this?

```
(define (repeated f n)
  (lambda (x) (f (f x))))
```

<sup>^</sup>That's not quite right, that just applies f twice. The goal is to apply fn times, as below:

```
(repeated f 1) ;; f(x)
(repeated f 2) ;; f(f(x))
(repeated f 3) ;; f(f(f(x)))
...
```

As a hint, notice how your solution takes n as input and yet doesn't do anything with it in the body - that's a good indicator that something's off.

iterative solution using double with log(n) time complexity

```
(define (repeated2-iter f n)
  (define (iter n current)
    (cond ((= 1 n) current)
          ((even? n) (double (iter (/ n 2) current)))
          (else (compose f (iter (- n 1) (compose f current)))))

    )
  (iter n f)
)
```

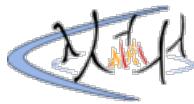
My own solution, that does not use compose. Should be considered silly.

```
(define (repeated f n)
  (if (= n 1)
      (lambda (x) (f x))
      (lambda (x)
        (f ((repeated f (- n 1)) x)))))

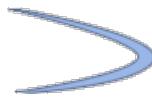
((repeated square 2) 5)
```

---

Last modified : 2022-12-30 16:29:19  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-1.44



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (1.43) | sicp-solutions | Next exercise (1.45) >>

```
(define dx 0.00001)

(define (smooth f)
  (lambda (x)
    (/ (+ (f (- x dx))
           (f x)
           (f (+ x dx)))
        3)))

(define (n-fold-smooth f n)
  ((repeated smooth n) f))
```

The input needs only the function and the number of smoothing procedure, so it should not include x.

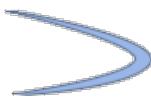
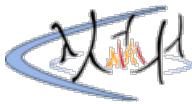
Be aware below is **WRONG**:

```
(define (n-fold-smooth f n)
  (repeated (smooth f) n))
```

This is illustrated below for  $n=2$

```
;; goal: n=2 -> "smooth the smoothed function"
((repeated smooth 2) f) ;; -> smooth(smooth(f)) [Correct]
(repeated (smooth f) 2) ;; -> smooth(f(smooth(f))) [Incorrect]
```

Last modified : 2022-05-15 20:35:33  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



<< Previous exercise (1.44) | Index | Next exercise (1.46) >>

Maggyero

## QUESTION

We saw in section 1.3.3 that attempting to compute square roots by naively finding a fixed point of  $y \mapsto x/y$  does not converge, and that this can be fixed by average damping. The same method works for finding cube roots as fixed points of the average-damped  $y \mapsto x/y^2$ . Unfortunately, the process does not work for fourth roots—a single average damp is not enough to make a fixed-point search for  $y \mapsto x/y^3$  converge. On the other hand, if we average damp twice (i.e., use the average damp of the average damp of  $y \mapsto x/y^3$ ) the fixed-point search does converge. Do some experiments to determine how many average damps are required to compute nth roots as a fixed-point search based upon repeated average damping of  $y \mapsto x/y^{n-1}$ . Use this to implement a simple procedure for computing nth roots using fixed-point, average-damp, and the repeated procedure of exercise 1.43. Assume that any arithmetic operations you need are available as primitives.

## ANSWER

```
(import (scheme small))

(define (nth-root x n)
  (define (fixed-point f first-guess)
    (define (try guess)
      (let ((next (f guess)))
        (if (close-enough? guess next)
            guess
            (try next))))
    (define (close-enough? x y)
      (< (abs (- x y)) 0.00001))
    (try first-guess)))
  (define (average-damp f)
    (lambda (x) (/ (+ x (f x)) 2)))
  (define (repeated f n)
    (define (compose f g)
      (lambda (x) (f (g x))))
    (if (= n 1)
        f
        (compose f (repeated f (- n 1)))))
  (fixed-point
    ((repeated average-damp (floor (log n 2)))
     (lambda (y) (/ x (expt y (- n 1))))))
    1.0))

; Average damping of  $y \mapsto x/y^{n-1}$  must be repeated  $\lfloor \log_2(n) \rfloor$  times to compute nth
roots of x as a fixed-point search.

(display (nth-root 2.0 2))
(newline)
(display (nth-root 2.0 3))
(newline)
(display (nth-root 2.0 4))
(newline)
(display (nth-root 2.0 5))
(newline)
(display (nth-root 2.0 6))
(newline)
(display (nth-root 2.0 7))
(newline)
(display (nth-root 2.0 8))
(newline)
```

```
(define (average x y)
  (/ (+ x y) 2.0))

(define (average-damp f)
  (lambda (x) (average x (f x))))
```

```

(define tolerance 0.00001)

(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))

(define (repeated f n)
  (if (= n 1)
      f
      (lambda (x) (f ((repeated f (- n 1)) x)))))

(define (get-max-pow n)
  (define (iter p r)
    (if (< (- n r) 0)
        (- p 1)
        (iter (+ p 1) (* r 2)))))

  (iter 1 2))

(define (pow b p)
  (define (even? x)
    (= (remainder x 2) 0))

  (define (sqr x)
    (* x x))

  (define (iter res a n)
    (if (= n 0)
        res
        (if (even? n)
            (iter res (sqr a) (/ n 2))
            (iter (* res a) a (- n 1))))))

  (iter 1 b p))

(define (nth-root n x)
  (fixed-point ((repeated average-damp (get-max-pow n))
                (lambda (y) (/ x (pow y (- n 1))))))
               1.0))

```

Example: (nth-root 5 32)

2.000001512995761

The number of times to repeat average-damp can also be calculated using floor and log to the base 2 as follows

```

(define (log2 x) (/ (log x) (log 2)))
(define (nth-root n x)
  (fixed-point ((repeated average-damp (floor (log2 n)))
                (lambda (y) (/ x (pow y (- n 1))))))
               1.0))

```

Kaihao

I think the above number of average damps required to compute nth root is wrong.

```

(define tolerance 0.00001)

(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))

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

(define (average-damp f)
  (lambda (x) (average x (f x))))

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

```

```

(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))

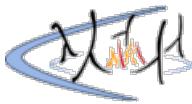
(define (compose f g)
  (lambda (x)
    (g (f x)))))

(define (repeated f n)
  (if (> n 1)
      (compose (repeated f (- n 1)) f)
      f))

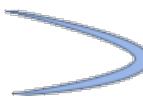
;; avarage-damp d times
(define (nth-root x n d)
  (fixed-point ((repeated average-damp d)
                (lambda (y) (/ x (fast-expt y (- n 1))))))
  1.0))

;; Experimental results from 2rd to 20th root:
;; Root Required Average Damps
;; 2 1
;; 3 1
;; 4 2
;; 5 2
;; 6 1
;; 7 1
;; 8 1
;; 9 1
;; 10 1
;; 11 1
;; 12 1
;; 13 3
;; 14 1
;; 15 1
;; 16 1
;; 17 1
;; 18 1
;; 19 1
;; 20 1

```



# sicp-ex-1.46



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (1.45) | Index | Next exercise (2.1) >>

Note that the arguments to *iterative-improve* must:

1. tell if a guess is **good enough**
2. improve a guess

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

(define (close-enough? v1 v2)
  (< (abs (- v1 v2)) tolerance))

(define (fixed-point f first-guess)
  ((iterative-improve
    (lambda (x) (close-enough? x (f x)))
    f)
   first-guess))

(define (sqrt x)
  ((iterative-improve
    (lambda (y)
      (< (abs (- (square y) x))
          0.0001))
    (lambda (y)
      (average y (/ x y))))
   1.0))
```

Here is a cleaner version of *iterative-improve* that reduces the number of state variables from three to one by using an inner procedure *iter*.

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

You can make it even cleaner by simply returning the inner procedure

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

You can make it even cleaner by removing the unnecessary lambda

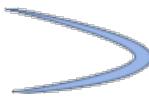
```
(define (iterative-improve good-enough? improve)
  (define (iter guess)
    (if (good-enough? guess)
        guess
        (iter (improve guess))))
  iter)
```

- The use of inner procedure avoids call of `iterative-improve` in second and after recursion.

- And the inner procedure captures `good-enough?` `improve` variables in `iterative-improve` thus return inner procedure is just ok.
- And the name of inner procedure is just for referencing itself in recursion.



# sicp-ex-2.1



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (1.46) | Index | Next exercise (2.2) >>

```
;; ex 2.1

(define (numer x) (car x))

(define (denom x) (cdr x))

(define (print-rat x)
  (newline)
  (display (numer x))
  (display "/")
  (display (denom x)))

(define (make-rat n d)
  (let ((g ((if (< d 0) - +) (gcd n d))))
    (cons (/ n g) (/ d g)))

;; Testing
(print-rat (make-rat 6 9)) ; 2/3
(print-rat (make-rat -6 9)) ; -2/3
(print-rat (make-rat 6 -9)) ; -2/3
(print-rat (make-rat -6 -9)) ; 2/3
```

There is a bug in the solution above. If gcd is defined as described in 1.2.5, it will have sign depending on the number of iterations it runs and the signs of a and b. For example:

```
(gcd 1 -2) ; 1
(gcd 6 -9) ; -3
```

Thus:

```
(print-rat (make-rat 1 -2)) ; 1/-2
(print-rat (make-rat 6 -9)) ; -2/3
```

To fix either make gcd return an absolute value or get the absolute value in make-rat (this is implemented below):

```
(define (make-rat n d)
  (let ((g ((if (< d 0) - +) (abs (gcd n d)))))
    (cons (/ n g) (/ d g))))
```

**category-learning-scheme category-texts**

Comment on how the above solution works:

n	d	g	n/g	d/g	rat
+	+	+	+	+	+
-	+	+	-	+	-
+	-	-	+	-	-
-	-	-	+	+	+

If we don't want to consider the sign of gcd, and still get the right answer, we could implement the following:

```
(define (make-rat n d)
  (define g (gcd n d))
  (cond ((> (* n d) 0) (cons (abs (/ n g)) (abs (/ d g))))
        ((< (* n d) 0) (cons (- (abs (/ n g))) (abs (/ d g))))))
```

Lily X. :)

We could save the simplified numbers as variables to make the program more readable too

```
(define (make-rat n d)
  (define g (gcd n d))
  (let ((simple-n (abs (/ n g)))
        (simple-d (abs (/ d g))))
    (cond
      ((> (* n d) 0) (cons simple-n simple-d))
      ((< (* n d) 0) (cons (- simple-n) simple-d)))))
```

Another solution follows:

```
(define (make-rat n d)
  (define (sign x) (if (< x 0) - +))
  (let ((g (gcd n d)))
    (cons ((sign d) (/ n g))
          (abs (/ d g)))))
```

Evan

Here's a solution that doesn't use conditionals.

```
(define (make-rat n d)
  (let ((g (abs (gcd n d)))
        (abs-d (abs d)))
    (cons (/ (* n (/ d abs-d)) g)
          (/ abs-d g))))
```

Let g take the absolute value of the gcd expression, so its sign won't interfere with the final sign of n and d.

Then, when you give cons, as its first argument, the multiplication of n by (/ d abs-d), you get the following:

- Numerator becomes negative if it was originally positive AND D is negative;
- Numerator remains negative if it was originally negative AND D is positive;
- Numerator remains positive if it was originally positive AND D is also positive;
- Numerator becomes positive if it was originally negative AND D is also negative;

Giving abs-d to cons as its second argument removes the possible sign from the denominator.

I really don't see why all the solutions here should look as cluttered as they do.

We just define sign as

```
(define (bool->number x)
  (cond ((= x #t) 1)
        ((= x #f) 0)
        (else "Input must be a boolean!"))
  )

(define (sign x)
  (- (bool->number (> x 0))
     (bool->number (< x 0)))
  )
```

And then make-rat as

```
(define (make-rat n d)
  (if (= d 0) (error "The denominator must not be zero!"))

  (define (rat-change x)
    (/ (* x (sign d)) (gcd n d)))
  )

  (cons (rat-change n) (rat-change d)))
)
```

Entropy Donary

A little redundant but I think very readable:

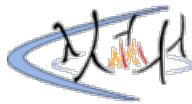
```
(define (make-rat n d)
  (if (< d 0)
```

```
(make-rat (- n) (- d))
(let ((g (abs (gcd n d))))
  (cons (/ n g) (/ d g)))))
```

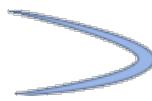
master

I also agree that the solutions here are generally too complicated. There are only two cases: The denominator is negative, in which case you should flip the sign of both the numerator and denominator, or the denominator is positive, in which case you do nothing. The solution above recognizes this fact but the solution is still very roundabout, although in a way sort of clever. Here's a straightforward solution:

```
(define (make-rat n d)
  (let ((g (gcd n d)))
    (if (negative? d)
        (cons (/ (- n) g) (/ (- d) g))
        (cons (/ n g) (/ d g)))))
```



# sicp-ex-2.2



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.1) | Index | Next exercise (2.3) >>

```
; ; ex 2.2. Straightforward

;; Point
(define (make-point x y) (cons x y))
(define (x-point p) (car p))
(define (y-point p) (cdr p))
(define (print-point p)
  (newline)
  (display "(")
  (display (x-point p))
  (display ",")
  (display (y-point p))
  (display ")"))

;; Segment
(define (make-segment start-point end-point)
  (cons start-point end-point))
(define (start-segment segment) (car segment))
(define (end-segment segment) (cdr segment))

(define (midpoint-segment segment)
  (define (average a b) (/ (+ a b) 2.0))
  (let ((a (start-segment segment))
        (b (end-segment segment)))
    (make-point (average (x-point a)
                         (x-point b))
                (average (y-point a)
                         (y-point b)))))

;; Testing
(define seg (make-segment (make-point 2 3)
                           (make-point 10 15)))

(print-point (midpoint-segment seg))
```

jz

Straightforward. Coming from Java/Ruby/C++, I can't say I care for the data being shoved arbitrarily into a list, but I guess that's why the layered approach and strict enforcement of using accessors like x-point is necessary.

lackita

I think the above solution misses part of the point about abstraction barriers; midpoint-segment reaches through both layers to achieve its goal.

```
(define (average-points a b)
  (make-point (average (x-point a) (x-point b))
              (average (y-point a) (y-point b)))))

(define (midpoint-segment seg)
  (average-points (start-segment seg)
                 (end-segment seg)))
```

muggy

Actually, I don't think the first solution violates abstraction barriers at all. It's almost completely analogous to the rational arithmetic procedures shown in the original text. Segment is directly above points in the abstraction ladder, so it's natural that segment uses point interface procedures like make-point, x-point, etc. The "average-points" procedure in your solution is simply another procedure added alongside make-point, etc. It improves code because the package directly above points can just use average-points without writing out the whole thing again and again.

In other words, it solves the problem of code repetition, and is quite different from the problem of abstraction barriers.

rohitkg98

Wrote a solution to demo real-world abstractions.

```
; An overview of how a real-world library would abstract details
; We wrap cons in make-point as a way to document that
; first argument is x, second is y
(define (make-point x y)
  (cons x y))

;x-point and y-point can be aliased to car and cdr because we don't lose any info
(define x-point car)

(define y-point cdr)

(define (print-point p)
  (newline)
  (display "(")
  (display (x-point p))
  (display ", ")
  (display y-point p)
  (display ")"))

; Abstracting out scalar operations on points with numbers
(define (apply-scalar-op op p val)
  (make-point (op (x-point p) val)
              (op (y-point p) val)))

; Building scalar ops using our abstraction
(define (divide-point p divisor)
  (apply-scalar-op / p divisor))

; Abstracting out application of scalar operation b/w points
(define (apply-scalar-op-on-points op p1 p2)
  (make-point (op (x-point p1) (x-point p2))
              (op (y-point p1) (y-point p2)))))

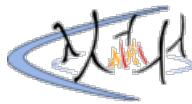
; Building scalar ops using our abstraction
(define (add-points p1 p2)
  (apply-scalar-op-on-points + p1 p2))

; Again, maintaining name of the arguments here
(define (make-segment start-point end-point)
  (cons start-point end-point))

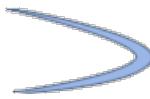
(define start-segment car)

(define end-segment cdr)

; Defining midpoint using our existing abstraction
(define (midpoint-segment segment)
  (divide-point (add-points (start-segment segment)
                            (end-segment segment))
                2.0))
```



# sicp-ex-2.3



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.2) | Index | Next exercise (2.4) >>

```
; ex 2.3. Not bothering with error/sanity checking.

;; Point
(define (make-point x y) (cons x y))
(define (x-point p) (car p))
(define (y-point p) (cdr p))

;; Rectangle - 1st implementation

(define (make-rect bottom-left top-right)
  (cons bottom-left top-right))

;; "Internal accessors", not to be used directly by clients. Not sure
;; how to signify this in scheme.
(define (bottom-left rect) (car rect))
(define (bottom-right rect)
  (make-point (x-point (cdr rect))
             (y-point (car rect))))
(define (top-left rect)
  (make-point (x-point (car rect))
             (y-point (cdr rect))))
(define (top-right rect) (cdr rect))

(define (width-rect rect)
  (abs (- (x-point (bottom-left rect))
          (x-point (bottom-right rect)))))
(define (height-rect rect)
  (abs (- (y-point (bottom-left rect))
          (y-point (top-left rect)))))

;; Public methods.
(define (area-rect rect)
  (* (width-rect rect) (height-rect rect)))
(define (perimeter-rect rect)
  (* (+ (width-rect rect) (height-rect rect)) 2))

;; Usage:
(define r (make-rect (make-point 1 1)
                      (make-point 3 7)))
(area-rect r)
(perimeter-rect r)

;; -----

;; Alternate implementation of rectangle. Note that this would screw
;; up clients that call make-rect directly, since it uses a different
;; number of args and different arg meanings, but it's generally bad
;; form for clients to call constructors directly anyway, they should
;; call some kind of factory method (cf "Domain Driven Design").

;; assuming, not checking width, height > 0.
(define (make-rect bottom-left width height)
  (cons bottom-left (cons width height)))

(define (height-rect rect) (cdr (cdr rect)))
(define (width-rect rect) (car (cdr rect)))

;; area and perimeter ops remain unchanged. The internal methods from
;; the first implementation won't work now.

;; Usage for second implementation:
(define r (make-rect (make-point 1 1) 2 6))
(area-rect r)
(perimeter-rect r)

;; Alternative Implementation II
;; -----
```

```

;;
;; The above implementations are limited to rectangles that have sides
;; parallel to the major axes of the plane. This implementation generalizes
;; to allow all rectangles. Conveniently enough, you can still use the above
;; area and perimeter definitions. Abstraction barrier for the win!
;;
;; DO NOTE -- As above all sanity/error checking has been ignored. IRL, you
;; you would want to ensure that parallel sides are actually parallel, etc.

;; Helpful to have this
(define (square x) (* x x))

;; Point library
(define (make-point x y) (cons x y))
(define (x-point p) (car p))
(define (y-point p) (cdr p))
(define (point-dist p1 p2)
  (sqrt (+ (square (- (x-point p1) (x-point p2)))
            (square (- (y-point p1) (y-point p2))))))

;; Segment library
(define (make-segment p1 p2) (cons p1 p2))
(define (start-seg p) (car p))
(define (end-seg p) (cdr p))
(define (seg-len seg) (point-dist (start-seg seg)
                                   (end-seg seg)))

;; Rectangle library
(define (make-rect side parallel-side)
  (cons side parallel-side))
(define (sidel rect) (car rect))
(define (side2 rect) (cdr rect))
(define (side-legths rect)
  (cons (seg-len (sidel rect))
        (min (abs (point-dist (start-seg (sidel rect))
                               (start-seg (side2 rect))))
              (abs (point-dist (start-seg (sidel rect))
                               (end-seg (side2 rect)))))))

;; Same as above
(define (width-rect rect) (car (side-legths rect)))
(define (height-rect rect) (cdr (side-legths rect)))

;; Usage
(define r (make-rect (make-segment (make-point 0 1)
                                     (make-point 0 0))
                      (make-segment (make-point 1 0)
                                    (make-point 1 1)))))

;; As an alternative to this alternative, You can define your rectangles
;; as a pair of perpendicular segments:

(define (make-rect side perpendicular-side)
  (cons side perpendicular-side))
(define (side-legths rect)
  (cons (seg-len (sidel rect))
        (seg-len (side2 rect)))))

;; And everything should still work.

;; Thus we now have 4 representations for rectangles, all of which can use the
;; same area and perimeter functions.

```

jz

I'm not sure if it's a drawback that you can't have public/private methods for the rectangle object or not. Smalltalk doesn't have such things either, but many other languages do. Anyway, the above works fine.

cmp

This implementation does not allow arbitrary rectangles in the plane. It is restricted to ones with sides parallel to the major axes. I am working on one that allows cockeyed rectangles. Was this feature intentionally left out?

Basically, you can give two parallel sides, or two intersecting (in the case of a rectangle perpendicular) sides. You then need "accessors" to compute heights and widths. Thus giving us two more ways to implement rectangles with the same area and perimeter calculations.

```

;; Here's another one: This allows arbitrary rotated rectangles, and the representation is the
;; easiest in my opinion. The rectangle is represented by the "base" - i.e. the segment with 2
;; bottom points, and the left side. To keep it simple, the input is the base, and the "height" from
;; the base. Here height is in the direction perpendicular to the base, and not along Y-axis.

```

```

;; This doesn't require error-checking as these parameters can't go wrong (base and height) and
a rectangle is uniquely defined by them.

(define (perimeter-r r)
  (let ((width (width-r r))
        (height (height-r r)))
    (* 2 (+ height width)))

(define (area-r r)
  (let ((width (width-r r))
        (height (height-r r)))
    (* width height)))

(define (width-r r)
  (length-seg (base-seg r)))

(define (height-r r)
  (length-seg (left-side r)))

(define (length-seg seg)
  (let ((p1 (start-segment seg))
        (p2 (end-segment seg)))
    (let ((x1 (x-point p1))
          (y1 (y-point p1))
          (x2 (x-point p2))
          (y2 (y-point p2)))
      (sqrt (+ (square (- x1 x2))
                (square (- y1 y2))))))

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

(define (base-seg r)
  (car r))

(define (left-side r)
  (cdr r))

(define (make-rectangle base-seg height)
  (let ((p1 (start-segment base-seg))
        (p2 (end-segment base-seg)))
    (let ((x1 (x-point p1))
          (y1 (y-point p1))
          (x2 (x-point p2))
          (y2 (y-point p2)))
      (let ((theta (atan (/ (- y2 y1)
                            (- x2 x1)))))

        (let ((new-x (- x1 (* height (sin theta))))
              (new-y (+ y1 (* height (cos theta)))))
          (cons base-seg
                (make-segment
                  p1
                  (make-point new-x new-y)))))))

```

Here's two more implementations, which assume axis-aligned rectangles, but allow users to enter any two points (i.e. they don't require bottom-left/top-right points as input), and satisfy identical signatures:

```

;; Representation 1: (cons (bottom-left point) (top-right point))
(define (make-rect p1 p2)
  (let ((x1 (x-point p1))
        (x2 (x-point p2))
        (y1 (y-point p1))
        (y2 (y-point p2)))
    (cond ((and (< x1 x2) (< y1 y2)) (cons p1 p2))
          ((and (> x1 x2) (> y1 y2)) (cons p2 p1))
          ((and (< x1 x2) (> y1 y2)) (cons (make-point x1 y2) (make-point x2 y1)))
          (else (cons (make-point x2 y1) (make-point x1 y2)))))

(define (bottom-left r)
  (car r))

(define (top-right r)
  (cdr r))

;; Representation 2: (cons (bottom-left point) (cons width height))
(define (make-rect p1 p2)
  (let ((x1 (x-point p1))
        (x2 (x-point p2))
        (y1 (y-point p1))
        (y2 (y-point p2)))
    (let ((width (abs (- x1 x2)))
          (height (abs (- y1 y2))))
      (cond ((and (< x1 x2) (< y1 y2)) (cons p1 (cons width height)))
            ((and (> x1 x2) (> y1 y2)) (cons p2 (cons width height)))
            ((and (< x1 x2) (> y1 y2)) (cons (make-point x1 y2) (cons width height)))))
```

```

(else (cons (make-point x2 y1) (cons width height)))))

(define (bottom-left r)
  (car r))

(define (top-right r)
  (let ((x (x-point (car r)))
        (y (y-point (car r))))
        (w (car (cdr r))))
        (h (cdr (cdr r)))))
    (make-point (+ x w) (+ y h))))
```

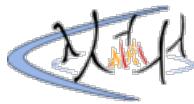
Either implementation can be used as follows:

```

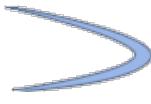
(define (print-rect r)
  (print-point (bottom-left r))
  (print-point (top-right r)))

(define (perimeter r)
  (let ((p1 (bottom-left r))
        (p2 (top-right r)))
    (let ((x1 (x-point p1))
          (x2 (x-point p2))
          (y1 (y-point p1))
          (y2 (y-point p2)))
      (* 2 (+ (- x2 x1) (- y2 y1)))))

(define (area r)
  (let ((p1 (bottom-left r))
        (p2 (top-right r)))
    (let ((x1 (x-point p1))
          (x2 (x-point p2))
          (y1 (y-point p1))
          (y2 (y-point p2)))
      (* (- x2 x1) (- y2 y1))))
```



# sicp-ex-2.4



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.3) | Index | Next exercise (2.5) >>

```
; ; ex 2.4. Alternate procedural rep of pairs.

; ; given:

(define (cons a b)
  (lambda (m) (m a b)))

; ; Commentary: cons returns a function that takes a function of 2
; ; args, a and b. The function will receive the values of a and b
; ; passed to cons when cons was called initially.

; ; z is a function that takes a 2-arg function. That inner function
; ; will be passed p and q in that order, so just return the first arg, p.
(define (car z)
  (z (lambda (p q) p)))

; ; ... so this is obvious.
(define (cdr z)
  (z (lambda (p q) q)))

; ; Usage:
(define x (cons 3 4))
(car x)
(cdr x)
```

Using applicative-order evaluation, verify that  $(\text{car } (\text{cons } x y))$  yields  $x$  for any objects  $x$  and  $y$ :

```
(car (cons x y))
(car (lambda (m) (m x y)))
((lambda (m) (m x y)) (lambda (p q) p))
((lambda (p q) p) x y)
x
```

jz

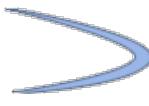
Proof that this works through the substitution rule:

```
(car (cons a b))
;-> ((cons a b) (lambda (p q) p))
;-> ((lambda (m) (m a b)) (lambda (p q) p))
;-> ((lambda (p q) p) a b)
;-> a
```

I'm not 100% with this demonstration, please edit if you can improve it. jz



# sicp-ex-2.5



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.4) | Index | Next exercise (2.6) >>

```
; ex 2.5, pairs of integers.

;; Pair a, b can be stored as a single integer  $2^a * 3^b$ , provided a
;; and b are both non-negative integers. The first part will be even,
;; the last odd. Getting rid of the even part will leave the odd, and
;; vice versa.

;; Helpers.

(define (exp base n)
  (define (iter x result)
    ; invariant:  $base^x * result$  is constant.
    (if (= 0 x)
        result
        (iter (- x 1) (* base result))))
  (iter n 1))

(define (count-0-remainder-divisions n divisor)
  (define (iter try-exp)
    (if (= 0 (remainder n (exp divisor try-exp)))
        (iter (+ try-exp 1)) ; Try another division.
        (- try-exp 1)))

  ; We don't need to try 0 divisions, as that will obviously pass.
  (iter 1))

;; cons, car, cdr
(define (my-cons a b) (* (exp 2 a) (exp 3 b)))
(define (my-car z) (count-0-remainder-divisions z 2))
(define (my-cdr z) (count-0-remainder-divisions z 3))

;; Usage:
(define test (my-cons 11 17))
(my-car test)
(my-cdr test)

;; Another way to define count-0-remainder-divisions

(define (divides? a b)
  (= 0 (remainder b a)))

(define (count-0-remainder-divisions n divisor)
  (define (iter x divisions)
    (if (divides? divisor x)
        (iter (/ x divisor) (+ divisions 1))
        divisions))
  (iter n 0))
```

jz

I had originally tried using an invariant for count-0-remainder-divisions, but couldn't get it to work. Alas.

atomik

The top example has a few more layers than are really necessary, so I flattened it out and added a printing interface to make testing easier.

```
; `count-0-remainder-divisions` will iteratively
; divide `p` by `d` until `p` is no longer a multiple
; of `d`. It will count the number of divisions with
; `n` and return `n` when `(remainder p d)` is not
; equal to 0.

(define (count-0-remainder-divisions n p d)
```

```

(if (= (remainder p d) 0)
    (count-0-remainder-divisions (+ n 1) (/ p d) d)
    n))

; Dr. Racket doesn't like it when I overwrite
; primitives so my functions will be named
; `pair` `head` and `tail` instead of `cons`
; `car` and `cdr`, respectively.

(define (pair a b)
  (* (expt 2 a) (expt 3 b)))

(define (head pair)
  (count-0-remainder-divisions 0 pair 2))

(define (tail pair)
  (count-0-remainder-divisions 0 pair 3))

(define (print pair)
  (newline)
  (display "(")
  (display (head pair))
  (display ".")
  (display (tail pair))
  (display ")"))
  (newline))

(print (pair 991 1023))

```

**chekkal**

```

;; Constructor
(define (cons a b) (* (expt 2 a) (expt 3 b)))
;; Enables us to calculate the log of n in base b
(define (logb b n) (floor (/ (log n) (log b))))
;; Selectors
(define (car x) (logb 2 (/ x (gcd x (expt 3 (logb 3 x))))))
(define (cdr x) (logb 3 (/ x (gcd x (expt 2 (logb 2 x))))))

```

**jstalton** This is probably similar to chekkal's answer at the end of the da<http://community.schemewiki.org/?sicp-ex-2.5y>. One downside is it returns floats, but this could probably be remedied via a round procedure (not yet introduced in course).

```

(define (cons a b)
  (* (expt 2 a) (expt 3 b)))

; once we factor out the 3s, then:
; 2^a = z
; log(2^a) = log(z)
; a*log(2) = log(z)
; a = log(z) / log (2)

(define (car z)
  (define (divisible-by-3? a)
    (= (remainder a 3) 0))
  (if (divisible-by-3? z)
      (car (/ z 3))
      (/ (log z) (log 2)))))

; once we factor out the 2s, then:
; 3^b = z
; log(3^b) = log(z)
; b * log(3) = log(z)
; b = log(z) / log(3)

(define (cdr z)
  (define (even? a)
    (= (remainder a 2) 0))http://community.schemewiki.org/?sicp-ex-2.5
  (if (even? z)
      (cdr (/ z 2))
      (/ (log z) (log 3)))))


```

**a00x** there exist some mistakes in chekkal's code, which leads to wrong results in some condition. For example, (cdr (cons 11 17)) will get 16 as output. Below are my codes,

```

(define (cons a b) (* (expt 2 a) (expt 3 b) ) )
(define (logb b n)
  (ceiling (/ (log n) (log b) ) )
)
)
(define (car n)
  (logb 2
        (gcd
          n
          (expt 2 (logb 2 n) ) )
        )
)
)
(define (cdr n)http://community.schemewiki.org/?sicp-ex-2.5
  (logb 3
        (gcd
          n
          (expt 3 (logb 3 n) ) )
        )
)
)
;test
(define x (cons 11 17) )
(car x)
(cdr x)

```

### Another implementation

foni

```

;;Helpers
;;Computes b^n ref: SICP -section 1.2.4
;; b^n = 1 if n = 0
;;      = (b^(n/2))^2 if n even
;;      = b.b^(n-1) o.w.
(define (expt b n)
  (define (even? n) (= (remainder n 2) 0))http://community.schemewiki.org/?sicp-ex-2.5
  (define (square x) (* x x))
  (cond ((= n 0) 1)
        ((even? n) (square (expt b (/ n 2))))
        (else (* b (expt b (- n 1))))))

;;return p where n = (k^p).q such that k does not divide q
(define (pow k n)
  (if (not (= 0 (remainder n k))) 0 (+ 1 (pow k (quotient n k)))))

;;test pow
(pow 2 (* (expt 2 3) (expt 7 11)))
(pow 7 (* (expt 2 3) (expt 7 11)))
(pow 5 (* (expt 2 3) (expt 7 11)))

;;;represent pairs of non-negative integers as 2^a.3^b
(define (cons a b) (* (expt 2 a) (expt 3 b)))
(define (car p) (pow 2 p))
(define (cdr p) (pow 3 p))

;;test the representation
(car (cons 12 34))
(cdr (cons 12 34))
(car (cons 3 0))
(cdr (cons 3 0))
(car (cons 0 3))
(cdr (cons 0 3))
(car (cons 0 0))
(cdr (cons 0 0))

```

### How about...

knucklehead

```

(define (cons x y)
  (* (expt 2 x) (expt 3 y)))

(define (car z)
  (if (= (gcd z 2) 1)
    0
    (+ 1 (car (/ z 2)))))


```

```
(define (cdr z)
  (if (= (gcd z 3) 1)
      0
      (+ 1 (cdr (/ z 3)))))
```

alexanderchr

The last solution can be abstracted further:

```
(define (largest-power-of a z)
  (if (= (remainder z a) 0)
      (+ 1 (largest-power-of a (/ z a)))
      0))

(define (car z)
  (largest-power-of 2 z))

(define (cdr z)
  (largest-power-of 3 z))
```

2bdkid

I turned alexanderch's function into a linear-iterative procedure.

```
(define (largest-power-divisor b n)
  (define (divides? a b)
    (= (remainder b a) 0))
  (define (iter result z)
    (if (divides? b z)
        (iter (inc result)
              (/ z b))
        result))
  (iter 0 n))

(define (num-cons a b)
  (* (expt 2 a)
     (expt 3 b)))

(define (num-car z)
  (largest-power-divisor 2 z))

(define (num-cdr z)
  (largest-power-divisor 3 z))
```

Nico de Vreeze

Using even?, / and cond.

```
(define (power base exp)
  (if (= exp 0)
      1
      (* base (power base (- exp 1)))))

(define (num-cons x y)
  (* (power 2 x) (power 3 y)))

;; and further only need even? If the number is not even and not 1, it must be divisible
by 3!

(define (num-car z)
  (cond ((= 1 z) 0)
        ((even? z) (+ 1 (num-car (/ z 2))))
        (else 0))) ; only factors of 3 left.

(define (num-cdr z)
  (cond ((= 1 z) 0)
        ((even? z) (num-cdr (/ z 2))) ; remove car-part.
        (else (+ 1 (num-cdr (/ z 3)))))) ; then divide by 3 until 1 left.

(define z (num-cons 10 20))
(num-car z) ;; 10
(num-cdr z) ;; 20
```

```
(define zeroes (num-cons 0 0))
(num-car zeroes) ;; 0
(num-cdr zeroes) ;; 0
```

adkipnis

Another implementation by iteratively finding the largest power divisor

```
; auxiliary procedures
(define (inc x) (+ 1 x))
(define (iter-div n d i)
  (if (= 0 (remainder n d))
      (iter-div (/ n d) d (inc i))
      i))

; constructors & selectors
(define (cons-e a b)
  (* (expt 2 a) (expt 3 b)))
(define (car-e z)
  (iter-div z 2 0))
(define (cdr-e z)
  (iter-div z 3 0))

; test case
(define z-e (cons-e 2 5))
(car-e z-e)
(cdr-e z-e)
```

mashomee

A clearer and simpler complete solution. though may be duplicated with some of above solutions.

```
(define (cons a b)
  (* (expt 2 a)
     (expt 3 b)))

(define (car z)
  (if (= (remainder z 2) 0)
      (1+ (car (/ z 2)))
      0))

(define (cdr z)
  (if (= (remainder z 3) 0)
      (1+ (cdr (/ z 3)))
      0))
```

master

I have a solution that's simple to understand. The idea is to repeatedly divide by 3 for the car or 2 for the cdr until the remainder is 0, which means the product doesn't have any of those factors anymore. Then take either the base 2 or base 3 logarithm to get the exponent of whatever is left over, which is the number we want. `car` and `cdr` have time complexity  $O(b)$  and  $O(a)$  respectively, and it's not obvious to me that you can do any better than that. As an added bonus, this solution is naturally tail-recursive so it also runs in constant space.

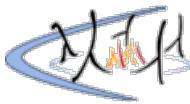
```
(define (logb base x)
  (/ (log x)
     (log base)))

(define (cons x y)
  (* (expt 2 x)
     (expt 3 y)))

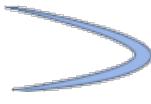
(define (car z)
  (if (= (remainder z 3) 0)
      (car (/ z 3))
      (logb 2 z)))

(define (cdr z)
  (if (= (remainder z 2) 0)
      (cdr (/ z 2))
      (logb 3 z)))
```





# sicp-ex-2.6



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.5) | Index | Next exercise (2.7) >>

The first tough question.

If zero is given as:

```
(define zero (lambda (f) (lambda (x) x)))
```

And add-1 is:

```
(define (add-1 n)
  (lambda (f) (lambda (x) (f ((n f) x)))))
```

one is the result of (add-1 zero)

```
(lambda (f) (lambda (x) (f ((zero f) x))))
```

zero is a function of one arg, that returns a function of one arg that returns the argument (identity function), so ((zero f) x) is just x. So, one reduces to:

```
(lambda (f) (lambda (x) (f x)))
```

two is the result of (add-1 one)

```
(lambda (f) (lambda (x) (f ((one f) x))))
```

substituting, and changing the lambda args for sanity:

```
=> (lambda (f) (lambda (x) (f (((lambda (a) (lambda (b) (a b))) f) x))))
=> (lambda (f) (lambda (x) (f ((lambda (b) (f b)) x))))
=> (lambda (f) (lambda (x) (f (f x))))
```

So, one and two are:

```
(define one (lambda (f) (lambda (x) (f x))))
(define two (lambda (f) (lambda (x) (f (f x)))))
```

Applying add-1 again for kicks gives

```
(define three (lambda (f) (lambda (x) (f (f (f x))))))
```

So, the function f (or whatever) gets applied again and again to the innermost x.

To define add generally, the result will just be the function applied n times, where n is the (numerical) sum. add needs to take 2 args, a and b, both of which are 2-level nested lambdas (as above). So, the method signature is:

```
(define (add a b) ...)
```

and add needs to return a 2-level nested lambda, as a and b will be:

```
(define (add a b)
  (lambda (f)
    (lambda (x)
      ...)))
```

Essentially, we need to apply a's multiple calls to its f (its outer lambda arg) to b's multiple calls to its f and its x. Poor explanation. We need the function calls to be like this:

```
(fa (fa (fa (fa ... (fa xa))))...)
```

and xa will be the call to b:

```
xa = (fb (fb (fb ... (fb x))) ...)
```

We pass f and x to b like this:

```
((b f) x)
```

so, we have

```
(fa (fa (fa (fa ... (fa ((b f) x))))))
```

but we need the function used by a in its outer lambda function to be the same as that used by b, or nothing will make any sense - using different functions would be like mixing octal and decimal numbers in a math problem. So, we pass f as the first lambda arg to a:

```
(a f)
```

and then a's inner lambda - a's x - will be ((b f) x)

So, the whole thing is:

```
(define (add a b)
  (lambda (f)
    (lambda (x)
      ((a f) ((b f) x)))))
```

Thanks to Ken Dyck for the hint.

Note: this explanation is brutal. If you are more articulate than I, let 'er rip.

jz

author of the ex 2.6 solution ^^

shyam

If you wonder how to use these functions here are few examples..

```
1 ]=> ((one square) 2)
;Value: 4

1 ]=> ((two square) 2)
;Value: 16

1 ]=> (((add two one) square) 2)
;Value: 256

1 ]=>
```

jsdalton

I found it helpful when experimenting to have a way to convert between integers and church numerals (e.g. to set up problems and also to confirm a solution). I used the following:

```
(define zero (lambda (f) (lambda (x) x)))

(define (add-1 n)
  (lambda (f) (lambda (x) (f ((n f) x)))))

(define (int-to-church n)
  (define (iter a result)
    (if (> a n)
        zero
        (add-1 (iter (+ a 1) result))
      ))
  (iter 1 zero))

(define (church-to-int cn)
  ((cn (lambda (n) (+ n 1))) 0))
```

anonymous

Hey jsdalton, thank you for the idea of procedures to convert between integers and church numerals. However, your int-to-church procedure doesn't need to define a helper procedure. The int-to-church procedure can be more succinctly written as:

```

;; Recursive process
(define (int->church n)
  (if (= n 0)
      zero
      (add-1 (int->church (- n 1)))))

;; Iterative process
(define (int->church n)
  (define (iter i result)
    (if (= i n)
        result
        (iter (+ i 1) (add-1 result))))
  (iter 0 zero))

```

anonymous

I think I have a much simpler and easier to understand solution, but I'm not convinced it's entirely correct

```
(define (add a b)
  ((a add-1) b))
```

Basically, you want to add-1 a times to b so you call a with add-1 and then call the resulting function with b to get a new function

Note: you should notice that this question says "not in terms of repeated application of add-1"

alexh

Here's yet another way of looking at it. A Church numeral is a second-order function: the numeral n is the function that turns f into "f composed with itself n times". Therefore we can define:

```
(define one (lambda (f) f))
(define two (lambda (f) (compose f f)))
```

where compose is the function from ex1.42:

```
(define (compose f g) (lambda (x) (f (g x))))
```

My short-cut for the "int->church" procedure is:

```
(define (print-church n)
  (display ((n inc) 0)) (newline))
```

Then we can do:

```
(define (church-add m n) (lambda (f) (compose (m f) (n f))))
(print-church (church-add one two))
```

Another interesting thing: multiplying Church numerals is actually easier than adding them!

```
(define (church-mult m n) (compose m n))
(define three (add-1 two))
(print-church (church-mult two three))
```

seok

An easier explanation to understand:

```

(define zero
  (lambda (f) (lambda (x) x)))
(define one
  (lambda (f) (lambda (x) (f x))))
(define two
  (lambda (f) (lambda (x) (f (f x)))))

#|
It can be easily proved that (n f) is (lambda (x) (f (f .. (f x) .. ))) where f is
repeated n times, by induction.

```

$$\begin{aligned}
 ((\text{add } n \text{ } m) \text{ } f) &= ((n+m) \text{ } f) \\
 &= (\lambda(x) (f^{(n+m)} \text{ } x)) \\
 &= (\lambda(x) ((\lambda(x) (f^n \text{ } x)) \text{ } (f^m \text{ } x))) \\
 &= (\lambda(x) ((\lambda(x) (f^n \text{ } x)) \text{ } ((\lambda(x) (f^m \text{ } x)) \text{ } x))) \\
 &= (\lambda(x) ((n \text{ } f) \text{ } ((m \text{ } f) \text{ } x)))
 \end{aligned}$$

```
(define (add n m)
  (lambda (f) (lambda (x) ((n f) ((m f) x)))))
```

yc

Maybe a duplicate, but still...

```
(define zero
  (lambda (f) (lambda (x) x)))
(define one
  (lambda (f) (lambda (x) (f x))))
(define two
  (lambda (f) (lambda (x) (f (f x)))))

;; with the above, we can get
(define n
  (lambda (f) (lambda (x) (f (f (f (f ... (f x))))))))
;; -----n times-----

;; examine `add-1'
(define (add-1 n)
  (lambda (f)
    (lambda (x)
      (f ((n f) x)))))

;; consider `((n f) x)' in `add-1'
((n f) x)
= (((lambda (f)
        (lambda (x)
          (f (f (f (f ... (f x)))))))
      f) x)
= ((lambda (x)
        (f (f (f (f ... (f x))))))
  x)
= (f (f (f (f ... (f x)))))

;; that is, f is applied n-times to x.
;; thus, in `add-1', one is added by prepending
;; ((n f) x) with an additional f
(f ((n f) x))

;; now, assume the result of applying f n-times
;; to x is n-result
(define n-result ((n f) x))

;; if we want to apply f another m-times to n-result
;; we would write
((m f) n-result)

;; which is
((m f) ((n f) x))

;; put the result back into `add-1', we get
(define (add-m-n m n)
  (lambda (f)
    (lambda (x)
      ((m f) ((n f) x)))))
```

# sicp-ex-2.7

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.6) | Index | Next exercise (2.8) >>

jz

```
;; ex 2.7
(define (make-interval a b) (cons a b))
(define (upper-bound interval) (max (car interval) (cdr interval)))
(define (lower-bound interval) (min (car interval) (cdr interval)))

;; Usage
(define i (make-interval 2 7))
(upper-bound i)
(lower-bound i)

(define j (make-interval 8 3))
(upper-bound j)
(lower-bound j)
```

We could create a function to get rid of the brief duplication in upper-bound and lower-bound, but it's not worth it.

mbsmith

The above is what I had as well; Though I think using min/max may not be necessary. If you look at the passage in the book regarding the add-interval procedure; The passage and the implementation indicate that lower-bound is positional rather than value dependent. At least as long as the make-interval procedure is unchanged.

Alyssa first writes a procedure for adding two intervals. She reasons that the minimum value the sum could be is the sum of the two lower bounds and the maximum value it could be is the sum of the two upper bounds:

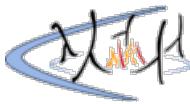
```
(define (add-interval x y)
  (make-interval (+ (lower-bound x) (lower-bound y))
                (+ (upper-bound x) (upper-bound y))))
```

So it looks like we could possibly define this as the following and still be within the constraints given by the book.

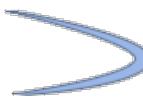
```
(define upper-bound cdr)
(define lower-bound car)
```

HannibalZ

I don't agree with Smith. Car and cdr work perfectly fine if the user always puts either upper-bound or lower-bound first, but I think the problem is that the user might put upper-bound first sometimes and put lower-bound first other times. In this case, interval data structure won't work. So I vote for using max and min.



# sicp-ex-2.8



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.7) | Index | Next exercise (2.9) >>

jz

; ; ex 2.8

```
; ; The max and min can be supplied to the constructor in any order.  
(define (make-interval a b) (cons a b))  
(define (upper-bound interval) (max (car interval) (cdr interval)))  
(define (lower-bound interval) (min (car interval) (cdr interval)))  
  
; ; The minimum value would be the smallest possible value  
; ; of the first minus the largest of the second. The maximum would be  
; ; the largest of the first minus the smallest of the second.  
(define (sub-interval x y)  
  (make-interval (- (lower-bound x) (upper-bound y))  
                (- (upper-bound x) (lower-bound y))))  
  
(define (display-interval i)  
  (newline)  
  (display "[")  
  (display (lower-bound i))  
  (display ",")  
  (display (upper-bound i))  
  (display "]"))  
  
; ; Usage  
(define i (make-interval 2 7))  
(define j (make-interval 8 3))  
  
(display-interval i)  
(display-interval (sub-interval i j))  
(display-interval (sub-interval j i))
```

shyam

Specifying subtraction in terms of addition is more accurate.

We also have the example of division specified in terms of multiplication in the text

```
(define (sub-interval x y)  
  (add-interval x (make-interval (- (upper-bound y)) (- (lower-bound y)))))
```

Where add-interval as given in the text is :

```
(define (add-interval x y)  
  (make-interval (+ (lower-bound x) (lower-bound y))  
                (+ (upper-bound x) (upper-bound y))))
```

This even works for negative valued interval, whether or not , we want to use values in that range ;-)

Here are some examples:

```
1 ]=> (sub-interval (make-interval 4 5) (make-interval 1 2))  
;Value 27: (2 . 4)  
  
1 ]=> (sub-interval (make-interval (- 4) (- 5)) (make-interval 1 2))  
;Value 28: (-7 . -5)  
  
1 ]=> (sub-interval (make-interval 4 5) (make-interval (- 1) 2))  
;Value 29: (2 . 6)  
  
1 ]=> (sub-interval (make-interval 4 5) (make-interval 1 (- 2)))  
;Value 30: (3 . 7)
```

```

1 ]=> (sub-interval (make-interval (- 4) 5) (make-interval 1 2))
;Value 31: (-6 . 4)

1 ]=> (sub-interval (make-interval 4 (- 5)) (make-interval 1 2))
;Value 32: (-7 . 3)

1 ]=>

```

dvdhsu

> This even works for negative valued interval, whether or not , we want to use values in that range ;-)

The proposed solution works for negative values as well. The minimum of two intervals is *always*  $(-\text{lower } a)(\text{upper } b)$ , and the maximum is *always*  $(-\text{upper } a)(\text{lower } b)$ .

Otherwise, though, you are correct; it certainly is more elegant to define subtraction in terms of addition.

rj1094

The above solutions are incorrect. To work correctly with ex 2.9, this width of the first interval minus the second interval must equal the width of the result of sub-interval.

So the resulting interval's lower bound is defined by subtracting the lower bounds of the inputs, and vice-versa for the upper bound:

```

(define (sub-interval x y)
  (make-interval (- (lower-bound x) (lower-bound y))
                (- (upper-bound x) (upper-bound y))))

```

rono

The above solutions are not incorrect, take a look at the following excerpt from ex 2.9:

Show that the width of the sum (or difference) of two intervals is a function only of the widths of the intervals being added (or subtracted)

Confusion might arise because it seems the exercise implies that:

- the sum of widths must be the widths of the intervals being added
- the difference of widths must be the widths of the intervals being subtracted

But that is not true, *added (or subtracted)* is being presented as a possibility which must be proven, rather than a correlation between the sum and the difference.

In any case, perhaps the following example can prove useful in understanding operations with intervals.

Subtracting two intervals **a** and **b** is the same as adding **a** and the **opposite of b** (as shown in the above solution by *shyam*), getting the opposite of an interval implies getting the opposites of its lower and upper bounds and swapping them.

Consider an interval with lower bound 1 and upper bound 3:[1, 3] Writing it as an inequality:

```
1 <= x <= 3
```

(x is any possible value in this range)

The opposite of this interval can be obtained by multiplying all of its members by **(-1)**

```

1(-1) <= x(-1) <= 3(-1)
-1 <= -x <= -3

```

This yields a untrue statement since x cant be simultaneously greater than **-1** and less than **-3**, this would require **-1 < -3** to be true, which is impossible since negative numbers get greater as they approach 0. This requires an additional step: the inequality relation must be reversed, and consequently, its lower and upper bounds will be reversed:

```
-1 <= -x <= -3 (impossible statement)
```

Reversed relations:

```
-1 >= -x >= -3
```

Which is the same as:

```
-3 <= -x <= -1
```

The opposite of the interval **[1, 3]** is **[-3, -1]**. Notice that adding **[-3, -1]** to another interval **a** would be the same as subtracting **[1, 3]** from **a** with the above solutions:

```
(sub-interval a [1, 3]) equals (add-interval a [-3, -1])
```

rj1094

The above solutions are incorrect. To work correctly with ex 2.9, this width of the first interval minus the second interval must equal the width of the result of sub-interval.

So the resulting interval's lower bound is defined by subtracting the lower bounds of the inputs, and vice-versa for the upper bound:

```
(define (sub-interval x y)
  (make-interval (- (lower-bound x) (lower-bound y))
                (- (upper-bound x) (upper-bound y))))
```

rono

The above solutions are not incorrect, take a look at the following excerpt from ex 2.9:

*Show that the width of the sum (or difference) of two intervals is a function only of the widths of the intervals being added (or subtracted)*

Confusion might arise because it seems the exercise implies that:

- the sum of widths must be the widths of the intervals being added
- the difference of widths must be the widths of the intervals being subtracted

But that is not true, *added (or subtracted)* is being presented as a possibility which must be proven, rather than a correlation between the sum and the difference.

In any case, perhaps the following example can prove useful in understanding operations with intervals.

Subtracting two intervals **a** and **b** is the same as adding **a** and the **opposite of b** (as shown in the above solution by *shyam*), getting the opposite of an interval implies getting the opposites of its lower and upper bounds and swapping them.

Consider an interval with lower bound 1 and upper bound 3:**[1, 3]** Writing it as an inequality:

```
1 <= x <= 3
```

(*x* is any possible value in this range)

The opposite of this interval can be obtained by multiplying all of its members by **(-1)**

```
1(-1) <= x(-1) <= 3(-1)
```

```
-1 <= -x <= -3
```

This yields a untrue statement since *x* cant be simultaneously greater than **-1** and less than **-3**, this would require **-1 < -3** to be true, which is impossible since negative numbers get greater as they approach 0. This requires an additional step: the inequality relation must be reversed, and consequently, its lower and upper bounds will be reversed:

```
-1 <= -x <= -3 (impossible statement)
```

Reversed relations:

```
-1 >= -x >= -3
```

Which is the same as:

```
-3 <= -x <= -1
```

The opposite of the interval **[1, 3]** is **[-3, -1]**. Notice that adding **[-3, -1]** to another interval **a** would be the same as subtracting **[1, 3]** from **a** with the above solutions:

```
(sub-interval a [1, 3]) equals (add-interval a [-3, -1])
```

rj1094

The above solutions are incorrect. To work correctly with ex 2.9, this width of the first interval minus the second interval must equal the width of the result of sub-interval.

So the resulting interval's lower bound is defined by subtracting the lower bounds of the inputs, and vice-versa for the upper bound:

```
(define (sub-interval x y)
  (make-interval (- (lower-bound x) (lower-bound y))
    (- (upper-bound x) (upper-bound y))))
```

rono

The above solutions are not incorrect, take a look at the following excerpt from ex 2.9:

*Show that the width of the sum (or difference) of two intervals is a function only of the widths of the intervals being added (or subtracted)*

Confusion might arise because it seems the exercise implies that:

- the sum of widths must be the widths of the intervals being added
- the difference of widths must be the widths of the intervals being subtracted

But that is not true, *added (or subtracted)* is being presented as a possibility which must be proven, rather than a correlation between the sum and the difference.

In any case, perhaps the following example can prove useful in understanding operations with intervals.

Subtracting two intervals **a** and **b** is the same as adding **a** and the **opposite of b** (as shown in the above solution by *shyam*), getting the opposite of an interval implies getting the opposites of its lower and upper bounds and swapping them.

Consider an interval with lower bound 1 and upper bound 3:[1, 3] Writing it as an inequality:

```
1 <= x <= 3
```

(x is any possible value in this range)

The opposite of this interval can be obtained by multiplying all of its members by(-1)

```
1 (-1) <= x (-1) <= 3 (-1)
-1 <= -x <= -3
```

This yields a untrue statement since x cant be simultaneously greater than-1 and less than-3, this would require **-1 < -3** to be true, which is impossible since negative numbers get greater as they approach 0. This requires an additional step: the inequality relation must be reversed, and consequently, its lower and upper bounds will be reversed:

```
-1 <= -x <= -3 (impossible statement)
```

Reversed relations:

```
-1 >= -x >= -3
```

Which is the same as:

```
-3 <= -x <= -1
```

The opposite of the interval [1, 3] is [-3, -1]. Notice that adding [-3, -1] to another interval **a** would be the same as subtracting [1, 3] from **a** with the above solutions:

```
(sub-interval a [1, 3]) equals (add-interval a [-3, -1])
```

Hammer

For those confused as to why when subtracting an interval you take a lower bound from x and an upper bound from y (and vice versa), consider that subtraction is addition with a negated second term; and that when a higher-bound is negated, it will become a lower-bound (and vice versa). To demonstrate, this intuitive solution produces equivalent results to (- lower-bound higher-bound) despite not specifying that.

```
;; ex 2.8
```

```
(define (lower-bound interval) (min (car interval) (cdr interval)))
(define (upper-bound interval) (max (car interval) (cdr interval)))
```

```
(define (negate-interval interval)
  (make-interval (* -1 (lower-bound interval)) (* -1 (upper-bound interval))))  
  
(define (add-interval x y)
  (make-interval (+ (lower-bound x) (lower-bound y))
                (+ (upper-bound x) (upper-bound y))))  
  
(define (sub-interval x y)
  (let ((y (negate-interval y)))
    (add-interval x y)))
```

---

Last modified : 2020-08-21 16:12:26  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-2.9

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

<< Previous exercise (2.8) | Index | Next exercise (2.10) >>

---

For addition and subtraction, the width of the result is a function of the widths of the input. For example,

$$[aL, aH] + [bL, bH] = [aL + bL, aH + bH].$$

*L = Low = lower bound, H = High = upper bound*

The width of this interval is

$$\begin{aligned} \text{width} &= 1/2 * ((aH + bH) - (aL + bL)) \\ &= 1/2 * ((aH - aL) + (bH - bL)) \\ &= \text{width of interval } a + \text{width of interval } b \end{aligned}$$

So, the width of the sum (or difference) of two intervals is just a function of the widths of those intervals.

For multiplication and division, the story is different. If the width of the result was a function of the widths of the inputs, then multiplying different intervals with the same widths should give the same answer. For example, multiplying a width 5 interval with a width 1 interval:

$$[0, 10] * [0, 2] = [0, 20] \quad (\text{width} = 10)$$

The following intervals have the same widths as the corresponding ones above, but multiplying gives different results:

$$[-5, 5] * [-1, 1] = [-5, 5] \quad (\text{width} = 5)$$

Thanks to jz for providing this solution.

# sicp-ex-2.10

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.9) | Index | Next exercise (2.11) >>

jz

Basically, we're just using the `error` keyword.

```
; ; ex 2.7
(define (make-interval a b) (cons a b))
(define (upper-bound interval) (max (car interval) (cdr interval)))
(define (lower-bound interval) (min (car interval) (cdr interval)))

(define (mul-interval x y)
  (let ((p1 (* (lower-bound x) (lower-bound y)))
        (p2 (* (lower-bound x) (upper-bound y)))
        (p3 (* (upper-bound x) (lower-bound y)))
        (p4 (* (upper-bound x) (upper-bound y))))
    (make-interval (min p1 p2 p3 p4)
                  (max p1 p2 p3 p4)))))

(define (div-interval x y)
  (mul-interval x
                (make-interval (/ 1. (upper-bound y))
                              (/ 1. (lower-bound y)))))

(define (print-interval name i)
  (newline)
  (display name)
  (display ": [")
  (display (lower-bound i))
  (display ",")
  (display (upper-bound i))
  (display "]"))

;; Usage
(define i (make-interval 2 7))
(define j (make-interval 8 3))

(print-interval "i" i)
(print-interval "j" j)
(print-interval "i*j" (mul-interval i j))
(print-interval "j*i" (mul-interval j i))
(print-interval "i/j" (div-interval i j))
(print-interval "j/i" (div-interval j i))

;; Gives:
;; i: [2,7]
;; j: [3,8]
;; i*j: [6,56]
;; j*i: [6,56]
;; i/j: [.25,2.333333333333333]
;; j/i: [.42857142857142855,4.]

;; New definition: Division by interval spanning 0 should fail.
;; Note that the operator <= has a different definition when using #lang racket and #lang sicp.
;; When using #lang sicp the operators to <= should be reversed.
(define (div-interval x y)
  (if (<= 0 (* (lower-bound y) (upper-bound y)))
      (error "Division error (interval spans 0)" y)
      (mul-interval x
                    (make-interval (/ 1. (upper-bound y))
                                  (/ 1. (lower-bound y))))))

(define span-0 (make-interval -1 1))
(print-interval "i/j" (div-interval i j))
(print-interval "i/span-0" (div-interval i span-0))

;; Results:
;; i/j: [.25,2.333333333333333]
;; ;Division error (interval spans 0) (-1 . 1)
```

vi

Couldn't we simply check if upper and lower bound are equal? Also, what happens here when dividing by an interval that starts at a negative and ends at a positive?

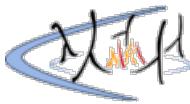
```
(div-interval (make-interval 0 1) (make-interval -2 2))
```

I propose this solution:

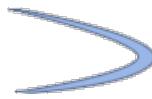
```
(define (div-interval x y)
  (if (= (upper-bound y) (lower-bound y))
      (error "division by zero interval")
      (mul-interval x
                    (make-interval (/ 1.0 (upper-bound y))
                                  (/ 1.0 (lower-bound y))))))
```

Tengs

The answer above(vi) misunderstand the question, if the interval is zero(upper-bound = lower-bound), that's alright, it won't cause any problem, the problem is the interval spans zero (upper-bound > 0, and lower-bound < 0), then if the dividend happens to be zero, it will cause problem. The first answer(jz) is correct.



# sicp-ex-2.11



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.10) | Index | Next exercise (2.12) >>

atomik

First things first, let's pre-empt some headaches by redefining our `make-interval` constructor.

```
(define (make-interval a b)
  (if (< a b)
      (cons a b)
      (cons b a)))
```

Now we can be certain that the lower-bound of any interval will be ALWAYS be less than its upper-bound. You can also place this test inside of the `upper-bound` and `lower-bound` procedures but since we'll be calling those functions more often than our constructor, I figure we should just put it in the constructor.

The next thing we want to do is define some kind of test to ensure correctness of our new `mul-interval` operation. This part took me longer than the rest of the exercise, but once I had some good tests set up I was able to get through the problem very quickly.

The tricky part was generating test data. I abused the `for-each` `map` and `append` procedures to make a procedure which takes two lists and produces a list of all intervals which could be produced from elements of each list. e.g. If you gave it '(a b) '(c d) it would give back '(ac ad bc bd). Because our interval constructor automatically orders the upper and lower bounds, (make-interval a c) and (make-interval c a) are equivalent.

```
; Warning. This is a hack. It makes no promises
; of performance, comprehension, or maintainability.
; It simply does what it needs to do.

(define (generate-intervals)
  (define test-list '())

  ; These lists can be edited by hand to produce different
  ; interval sets. I try to make sure both lists contain
  ; at least one 0 and some negative/positive numbers
  ; with same/different values to ensure a variety of
  ; intervals.

  (define test-data
    (cons (list 0 1 2 3 4 5 -6 -7 -8 -9 -10)
          (list 5 4 3 2 1 0 -1 -2 -3 -4 -5)))
  (for-each
    (lambda (x) (set! test-list (append test-list (list x))))
    (map (lambda (x) (map (lambda (y) (make-interval x y))
                           (cdr test-data)))
          (car test-data)))

  ; Our testing procedure will also be abusing for-each
  ; and map to make combinations, so we take our test list
  ; and pair it with its reverse ensure more varied
  ; combinations of pairs.

  (cons test-list (reverse test-list)))

; Capture the result of `generate-intervals` so we don't have to
; run it again.

(define test-intervals
  (generate-intervals))
```

Now that nasty business is over we can write out our test.

```
; `test` will take two procedures, call each one
; with the same data and compare their results.
; We will hard-code the test-data because the
; alternative is more than my job's worth. If
; you want to use a different data-set, alter
; the `generate-intervals` procedure.
```

```

(define (test f g)

  ; We need to define a special kind
  ; of equality operator for intervals

  (define (interval-equals a b)
    (and (= (lower-bound a) (lower-bound b)) (= (upper-bound a) (upper-bound
b)))))

  ; We will test every single possible combination
  ; of pairs from either list. Thanks to the
  ; commutativity of multiplication, this is fairly
  ; straightforward.
  ; If a pair passes the test, nothing gets printed
  ; to the console, but if a pair fails, then both
  ; the original intervals, as well as the results
  ; of applying f and g to said intervals will be printed.

  (for-each (lambda (x)
    (for-each (lambda (y)
      (cond ((interval-equals (f x) (g x y)) #t)
            (else
              (newline)
              (display "failed on inputs: ")
              (display x) (display y)
              (newline)
              (display (f x y)) (display (g x
y))
              (newline))))
      (cdr test-intervals)))
    (car test-intervals)))

```

We still need more tests, but nothing as awful as what we've already done. These next tests will be used in the body of our new mul-interval procedure. We want to define some procedures to tell us when a pair of numbers are both negative, both positive, or have opposite signs. I decided to start by testing for opposite-ness and then I defined the other two tests in terms of the opposite test.

```

(define (opposite-pair? a b)
  (if (positive? a)
    (negative? b)
    (positive? b)))

(define (positive-pair? a b)
  (if (opposite-pair? a b)
    #f
    (positive? a)))

(define (negative-pair? a b)
  (if (opposite-pair? a b)
    #f
    (negative? a)))

```

NOW we can start writing our new interval multiplication procedure, with the help of our `test` procedure. We will keep the old `mul-interval` procedure in scope of our new one, so we can call it during the course of designing our case analysis. This ensures that we're never breaking any more than we need to at any given time.

```

(define (old-mul-interval x y)
  (let ((p1 (* (lower-bound x) (lower-bound y))))
    (p2 (* (lower-bound x) (upper-bound y)))
    (p3 (* (upper-bound x) (lower-bound y)))
    (p4 (* (upper-bound x) (upper-bound y))))
  (make-interval
    (min p1 p2 p3 p4)
    (max p1 p2 p3 p4)))

(define (mul-interval x y)

  ; We will capture the boundaries
  ; of each interval as variables
  ; to avoid repeated function calls

  (let ((x0 (lower-bound x))
        (x1 (upper-bound x))
        (y0 (lower-bound y))
        (y1 (upper-bound y)))

    ; At the moment, mul-interval just
    ; passes its arguments on to
    ; `old-mul-interval`

    (cond (else (old-mul-interval x y)))))

; nothing will be printed as both procedures are basically the same.

```

```
(test old-mul-interval mul-interval)
```

Now we can design our cases one at a time. For instance:

```
(define (old-mul-interval x y)
  (let ((p1 (* (lower-bound x) (lower-bound y)))
        (p2 (* (lower-bound x) (upper-bound y)))
        (p3 (* (upper-bound x) (lower-bound y)))
        (p4 (* (upper-bound x) (upper-bound y))))
    (make-interval
      (min p1 p2 p3 p4)
      (max p1 p2 p3 p4)))))

(define (mul-interval x y)
  (let ((x0 (lower-bound x))
        (x1 (upper-bound x))
        (y0 (lower-bound y))
        (y1 (upper-bound y)))

    (cond ((negative-pair? x0 x1)
           (make-interval (* x0 x1) (* y0 y1)))
          (else (old-mul-interval x y)))))

(test old-mul-interval mul-interval)
```

The above code will only fail and print results for multiplications in which the first pair is negative. If we add a nested cond block we can narrow even further:

```
(define (old-mul-interval x y)
  (let ((p1 (* (lower-bound x) (lower-bound y)))
        (p2 (* (lower-bound x) (upper-bound y)))
        (p3 (* (upper-bound x) (lower-bound y)))
        (p4 (* (upper-bound x) (upper-bound y))))
    (make-interval
      (min p1 p2 p3 p4)
      (max p1 p2 p3 p4)))))

(define (mul-interval x y)
  (let ((x0 (lower-bound x))
        (x1 (upper-bound x))
        (y0 (lower-bound y))
        (y1 (upper-bound y)))

    (cond ((negative-pair? x0 x1)
           (cond ((negative-pair? y0 y1)
                  (make-interval (* x0 y0) (* x1 y1)))
                  (else (old-mul-interval x y)))))

          (else (old-mul-interval x y)))))

(test old-mul-interval mul-interval)
```

Now it will only fail and print results for multiplications in which the first pair is negative and the second pair is negative.

I'm gonna skip to the answer now. If you're reading this because you're stuck, I encourage you to take what I wrote, try to solve the rest of the problem and then come back.

```
(define (make-interval a b)
  (if (< a b)
      (cons a b)
      (cons b a)))

(define (lower-bound interval) (car interval))
(define (upper-bound interval) (cdr interval))

(define (mul-interval x y)
  (define (opposite-pair? a b)
    (if (positive? a)
        (negative? b)
        (positive? b)))

  (define (positive-pair? a b)
    (if (opposite-pair? a b)
        #f
        (positive? a)))

  (define (negative-pair? a b)
    (if (opposite-pair? a b)
        #f
        (negative? a)))
  (let ((x0 (lower-bound x))
        (x1 (upper-bound x))
        (y0 (lower-bound y))
        (y1 (upper-bound y))))
```

```

(y0 (lower-bound y))
(y1 (upper-bound y)))
(cond ((negative-pair? x0 x1)
       (cond ((opposite-pair? y0 y1)
              (make-interval (* x0 y0) (* x0 y1)))
              ((negative-pair? y0 y1)
               (make-interval (* x1 y1) (* x0 y0)))
              (else
               (make-interval (* x1 y0) (* x0 y1)))))
        ((positive-pair? x0 x1)
         (cond ((opposite-pair? y0 y1)
                (make-interval (* x1 y0) (* x1 y1)))
                ((negative-pair? y0 y1)
                 (make-interval (* x1 y0) (* x0 y1)))
                (else
                 (make-interval (* x0 y0) (* x1 y1)))))
        (else
         (cond ((positive-pair? y0 y1)
                (make-interval (* x0 y1) (* x1 y1)))
                ((negative-pair? y0 y1)
                 (make-interval (* x1 y0) (* x0 y0)))
                (else
                 (make-interval
                  ((lambda (a b) (if (< a b) a b))
                   (* x0 y1) (* x1 y0)))
                  ((lambda (a b) (if (> a b) a b))
                   (* x0 y0) (* x1 y1))))))))
(define (generate-intervals)
  (define test-list '())
  (define test-data
    (cons (list 0 1 2 3 4 5 -6 -7 -8 -9 -10)
          (list 5 4 3 2 1 0 -1 -2 -3 -4 -5)))
  (for-each
    (lambda (x) (set! test-list (append test-list x)))
    (map (lambda (x) (map (lambda (y) (make-interval x y))
                           (cdr test-data)))
         (car test-data)))
  (cons test-list test-list))

(define test-intervals
  (generate-intervals))

(define (test f g)
  (define (interval>equals a b)
    (and (= (lower-bound a) (lower-bound b)) (= (upper-bound a) (upper-bound b))))
  (for-each (lambda (x)
    (for-each (lambda (y)
      (cond ((interval>equals (f x) (g x y)) #t)
            (else
              (newline)
              (display x) (display y)
              (newline)
              (display (f x y)) (display (g x y))
              (newline))))
    (cdr test-intervals)))
  (car test-intervals)))

(define (old-mul-interval x y)
  (let ((p1 (* (lower-bound x) (lower-bound y))))
    (p2 (* (lower-bound x) (upper-bound y)))
    (p3 (* (upper-bound x) (lower-bound y)))
    (p4 (* (upper-bound x) (upper-bound y))))
  (make-interval
    (min p1 p2 p3 p4)
    (max p1 p2 p3 p4)))

(test old-mul-interval mul-interval)

```

The new mul-interval procedure is WAY longer than the old one. Maybe some day I'll come back to it and check to see if all this optimization was really worth it. Let me know if I made any mistakes. I know the testing procedures don't completely combine the sets (each set should be able to combine with itself). Plus I notice everyone else was fussing about intervals which span 0 or have 0 as one or both boundaries. I think those cases were automatically generated and thus covered in my tests but I could be wrong.

jared-ross

I agree with everyone else here, Ben is very mean, this took a long time to answer.

First I split intervals into three groups using this interval sign:

; Signs for Intervals:

```

; +1: Sits only in the positives
; 0: Contains 0, or includes 0 in it's bounds
; -1: Sits only in the negatives
;
; To see how this works draw a chart of the operator, along side a
; chart of s.
(define (sign-interval iv)
  (let* ((l (lower-bound-interval iv))
         (u (upper-bound-interval iv))
         (s (sign (* u l))))
    (if (= s -1)
        0
        (* (sign l) s))))

```

Then I went through every combination of sign of two intervals, discarding with respect commutating (when switching the signs around gives you the same result as something else you have worked out), on paper, working out the combinations of multiplications needed, I resulted with this:

```

(define (symbol->sign s)
  (cond
    ((eq? s 0) 0)
    ((eq? s '+) 1)
    ((eq? s '-) -1)))

; This took me a long time to do.
; I hope you appreciate this imagined reader of my code.
(define (mul-interval a b)
  (let ((sa (sign-interval a))
        (sb (sign-interval b)))
    ; Signs: Returns true if the signs given match the intervals
    (define (signs a b)
      (and (= sa (symbol->sign a)) (= sb (symbol->sign b))))
    ; Pos: Retrieves the (p)osition of the (i)nterval
    (define (pos i p)
      (if (eq? p 'u) (upper-bound-interval i) (lower-bound-interval i)))
    ; Mult: Multiplies the value of the position of a (pa) by the value of the position of
    ; b (pb)
    (define (mult pa pb) ; (u)pper or (l)ower for pa and pb
      (* (pos a pa) (pos b pb)))
    ; This is the core of the function:
    (cond
      ((signs '+ '+) (make-interval (mult 'l 'l) (mult 'u 'u)))
      ((signs '- '+) (make-interval (mult 'l 'u) (mult 'u 'l)))
      ((signs 0 '+) (make-interval (mult 'l 'u) (mult 'u 'u)))
      ((signs '+ '-) (mul-interval b a))
      ((signs '- '-) (make-interval (mult 'u 'u) (mult 'l 'l)))
      ((signs 0 '-) (make-interval (mult 'u 'l) (mult 'l 'l)))
      ((signs '+ 0) (mul-interval b a))
      ((signs '- 0) (mul-interval b a))
      ((signs 0 0) (make-interval
                     (min (mult 'l 'u) (mult 'u 'l))
                     (max (mult 'l 'l) (mult 'u 'u)))))
      )
    )
  )
)
```

Then I built some code to test it:

```

(define (random-interval)
  (define width 5)
  (define granularity 0.25)
  (define (random-point)
    (let* ((num-granules (inexact->exact (/ width granularity)))
           (a-granule (random (+ num-granules 1)))
           (scaled-granule (* a-granule granularity))
           (scaled-and-shifted-granule (- scaled-granule (/ width 2))))
           scaled-and-shifted-granule))
    (let ((p1 (random-point))
          (p2 (random-point)))
      (make-interval (min p1 p2) (max p1 p2)))))

(define (format-interval iv)
  (define $ number->string)
  (string-append "[" ($ (lower-bound-interval iv)) ", " ($ (upper-bound-interval iv)) "]"))

(define (repeat-call f n)
  (f)
  (if (= n 1)
      nil
      (repeat-call f (- n 1)))
)

(define (equal?-interval a b)

```

```

(and
(= 
  (upper-bound-interval a)
  (upper-bound-interval b))
(= 
  (lower-bound-interval a)
  (lower-bound-interval b)))))

(define (test-new-mul-interval)
(define number-of-tests 1000)
(define (error i1 i2)
  (newline)
  (display "Failed Match: ")
  (display (format-interval i1))
  (display ", ")
  (display (format-interval i2))
  (display " => ")
  (display (format-interval (old-mul-interval i1 i2))))
  (display " != ")
  (display (format-interval (mul-interval i1 i2)))
  (newline))
(define (test)
  (let ((i1 (random-interval))
        (i2 (random-interval)))
    (if
      (equal?-interval (old-mul-interval i1 i2) (mul-interval i1 i2))
      (display ".")
      (error i1 i2))
    )))
(repeat-call test number-of-tests)
)

```

And everything seems to pass, so I think it is working :)

Contact me at (join-with-dots jared b ross) on gmail

jz

Ben Bitdiddle should stick to diddling his own bits. His comment just obfuscates the code. But, if we know the signs of the endpoints, we have 3 possible cases for each interval: both ends positive, both negative, or an interval spanning zero; therefore, there are  $3 \times 3 = 9$  possible cases to test for, which reduces the number of multiplications, except when both intervals span zero.

```

;; ex 2.7
(define (make-interval a b) (cons a b))
(define (upper-bound interval) (max (car interval) (cdr interval)))
(define (lower-bound interval) (min (car interval) (cdr interval)))

(define (print-interval name i)
  (newline)
  (display name)
  (display ": [")
  (display (lower-bound i))
  (display ", ")
  (display (upper-bound i))
  (display "]"))

;; Old multiplication (given)
(define (old-mul-interval x y)
  (let ((p1 (* (lower-bound x) (lower-bound y))))
    (p2 (* (lower-bound x) (upper-bound y)))
    (p3 (* (upper-bound x) (lower-bound y)))
    (p4 (* (upper-bound x) (upper-bound y))))
  (make-interval (min p1 p2 p3 p4)
                (max p1 p2 p3 p4)))

;; This looks a *lot* more complicated to me, and with the extra
;; function calls I'm not sure that the complexity is worth it.
(define (mul-interval x y)
  ;; endpoint-sign returns:
  ;;   +1 if both endpoints non-negative,
  ;;   -1 if both negative,
  ;;   0 if opposite sign
  (define (endpoint-sign i)
    (cond ((and (>= (upper-bound i) 0)
                (>= (lower-bound i) 0))
           1)
          ((and (< (upper-bound i) 0)
                (< (lower-bound i) 0))
           -1)
          (else 0)))

```

```

(let ((es-x (endpoint-sign x))
      (es-y (endpoint-sign y))
      (x-up (upper-bound x))
      (x-lo (lower-bound x))
      (y-up (upper-bound y))
      (y-lo (lower-bound y)))

  (cond ((> es-x 0) ;; both x endpoints are +ve or 0
         (cond ((> es-y 0)
                 (make-interval (* x-lo y-lo) (* x-up y-up)))
               ((< es-y 0)
                 (make-interval (* x-up y-lo) (* x-lo y-up)))
               (else
                 (make-interval (* x-up y-lo) (* x-up y-up)))))

        ((< es-x 0) ;; both x endpoints are -ve
         (cond ((> es-y 0)
                 (make-interval (* x-lo y-up) (* x-up y-lo)))
               ((< es-y 0)
                 (make-interval (* x-up y-up) (* x-lo y-lo)))
               (else
                 (make-interval (* x-lo y-up) (* x-lo y-lo)))))

        (else ;; x spans 0
         (cond ((> es-y 0)
                 (make-interval (* x-lo y-up) (* x-up y-up)))
               ((< es-y 0)
                 (make-interval (* x-up y-lo) (* x-lo y-lo)))
               (else
                 ;; Both x and y span 0 ... need to check values
                 (make-interval (min (* x-lo y-up) (* x-up y-lo))
                               (max (* x-lo y-lo) (* x-up y-up)))))))

```

The above is so gross I tested it out. Yes, I'm a keener.

```

(define (eql-interval? a b)
  (and (= (upper-bound a) (upper-bound b))
       (= (lower-bound a) (lower-bound b)))

;; Fails if the new mult doesn't return the same answer as the old
;; naive mult.
(define (ensure-mult-works aH aL bH bL)
  (let ((a (make-interval aL aH))
        (b (make-interval bL bH)))
    (if (eql-interval? (old-mul-interval a b)
                       (mul-interval a b))
        true
        (error "new mult returns different value!"
              a
              b
              (old-mul-interval a b)
              (mul-interval a b)))))

;; The following is overkill, but it found some errors in my
;; work. The first two #'s are the endpoints of one interval, the last
;; two are the other's. There are 3 possible layouts (both pos, both
;; neg, one pos one neg), with 0's added for edge cases (pos-0, 0-0,
;; 0-neg).

(ensure-mult-works +10 +10 +10 +10)
(ensure-mult-works +10 +10 +00 +10)
(ensure-mult-works +10 +10 +00 +00)
(ensure-mult-works +10 +10 +10 -10)
(ensure-mult-works +10 +10 -10 +00)
(ensure-mult-works +10 +10 -10 -10)

(ensure-mult-works +00 +10 +10 +10)
(ensure-mult-works +00 +10 +00 +10)
(ensure-mult-works +00 +10 +00 +00)
(ensure-mult-works +00 +10 +10 -10)
(ensure-mult-works +00 +10 -10 +00)
(ensure-mult-works +00 +10 -10 -10)

(ensure-mult-works +00 +00 +10 +10)
(ensure-mult-works +00 +00 +00 +10)
(ensure-mult-works +00 +00 +00 +00)
(ensure-mult-works +00 +00 +10 -10)
(ensure-mult-works +00 +00 -10 +00)
(ensure-mult-works +00 +00 -10 -10)

(ensure-mult-works +10 -10 +10 +10)
(ensure-mult-works +10 -10 +00 +10)

```

```

(ensure-mult-works +10 -10 +00 +00)
(ensure-mult-works +10 -10 +10 -10)
(ensure-mult-works +10 -10 -10 +00)
(ensure-mult-works +10 -10 -10 -10)

(ensure-mult-works -10 +00 +10 +10)
(ensure-mult-works -10 +00 +00 +10)
(ensure-mult-works -10 +00 +00 +00)
(ensure-mult-works -10 +00 +10 -10)
(ensure-mult-works -10 +00 -10 +00)
(ensure-mult-works -10 +00 -10 -10)

(ensure-mult-works -10 -10 +10 +10)
(ensure-mult-works -10 -10 +00 +10)
(ensure-mult-works -10 -10 +00 +00)
(ensure-mult-works -10 -10 +10 -10)
(ensure-mult-works -10 -10 -10 +00)
(ensure-mult-works -10 -10 -10 -10)

;; All of these run without any errors now.

```

jsdalton

Yeah, Ben certainly has an evil streak.

I was able to reduce a bit of the clutter by reworking the way the conditional logic is framed. I observed that there were certain patterns in where values were getting passed to the call to make-interval. So rather than working through the conditions and calling make-interval accordingly, I set it up to select the appropriate value for each "slot".

For clarity everything else is the same as jz's solution above. The testing procedures he came up with were also invaluable in working out kinks. Ultimately I'm not sure my solution is any clearer or better -- it's quite possibly neither. It is a tiny bit more concise though:

```

(define (mul-interval x y)
  (define (endpoint-sign i)
    (cond ((and (>= (upper-bound i) 0)
                (>= (lower-bound i) 0))
           1)
          ((and (< (upper-bound i) 0)
                (< (lower-bound i) 0))
           -1)
          (else 0)))

  (let ((es-x (endpoint-sign x))
        (es-y (endpoint-sign y))
        (x-up (upper-bound x))
        (x-lo (lower-bound x))
        (y-up (upper-bound y))
        (y-lo (lower-bound y)))

    (if (and (= es-x 0) (= es-y 0))
        ; Take care of the exceptional condition where we have to test
        (make-interval (min (* x-lo y-up) (* x-up y-lo))
                      (max (* x-lo y-lo) (* x-up y-up)))
        ; Otherwise, select which value goes in which "slot". I'm not sure
        ; whether there is an intuitive way to explain *why* these
        ; selections work.
        (let ((a1 (if (and (<= es-y 0) (<= (- es-y es-x) 0)) x-up x-lo))
              (a2 (if (and (<= es-x 0) (<= (- es-x es-y) 0)) y-up y-lo))
              (b1 (if (and (<= es-y 0) (<= (+ es-y es-x) 0)) x-lo x-up))
              (b2 (if (and (<= es-x 0) (<= (+ es-x es-y) 0)) y-lo y-up)))
          (make-interval (* a1 a2) (* b1 b2)))))))

```

vpraid

This solution has 9 cases, exactly as it is required by the problem statement, and all of them are clearly visible (although I wouldn't say readable). I had to redefine positive? predicate that is provided by my scheme interpreter. It also passes all of jz's test cases.

```

(define (mul-interval x y)
  (define (positive? x) (>= x 0))
  (define (negative? x) (< x 0))
  (let ((xl (lower-bound x))
        (xu (upper-bound x))
        (yl (lower-bound y))
        (yu (upper-bound y)))
    (cond ((and (positive? xl) (positive? yl))

```

```

(make-interval (* xl yl) (* xu yu)))
((and (positive? xl) (negative? yl))
 (make-interval (* xu yl) (* (if (negative? yu) xl xu) yu)))
 ((and (negative? xl) (positive? yl))
 (make-interval (* xl yu) (* xu (if (negative? xu) yl yu))))
 ((and (positive? xu) (positive? yu))
 (let ((l (min (* xl yu) (* xu yl))))
 (u (max (* xl yl) (* xu yu)))))
 (make-interval l u)))
 ((and (positive? xu) (negative? yu))
 (make-interval (* xu yl) (* xl yl)))
 ((and (negative? xu) (positive? yu))
 (make-interval (* xl yu) (* xl yl)))
 (else
 (make-interval (* xu yu) (* xl yl))))))

```

jwilly

I believe jz's tests go beyond the scope of this problem by allowing an interval's lower bound to be greater than its upper bound and can make this problem seem more difficult than it actually is.

The constructor for make-interval given in the book (2nd edition) does not swap the values if the first parameter is greater than the second parameter. Instead the onus is on the client to use the function properly. The client should assume that the lower bound has to be smaller than the upper bound. jz's tests while all encompassing will not pass vpraid's solution unless you alter the make-interval constructor to swap out of order values.

rjk

This was a mess, and I agree that Ben is extremely unhelpful. I worked out the nine combinations of legal interval signs (eg, an interval cannot be +- because any positive number is greater than any negative. Having worked out those combinations, I generated some test intervals to see what the max and min products for each possible combination were. (I actually did this in python, as im much more fluent in that, and wasn't really sure how to do it in scheme. atomik did it up top, but used for-each and map, but since I'm working through the book in order, i haven't got to those yet).

This got me which bounds to multiply for the min and max for each. I then realized I had the problem of zeros. So, I systematically tested effects of a zero as an endpoint for each type of interval (++,-+,--) and came to the conclusion that i could treat zeros as positive, as they would behave in that way, or push the multiplication into the trouble case of -+\*-+.

What followed was the cond block to end all cond blocks. My next challenge would be to make a procedure that simplifies those ands at least, but I want to move on.

```

; patt | min | max
; ++++ | al bl | ah bh
; +--+ | ah bl | ah bh
; +-+ | ah bl | al bh
; -++ | al bh | ah bh
; -+- | trouble case
; --- | ah bl | al bl
; --+ | al bh | ah bl
; ---+ | al bh | al bl
; ---- | ah bh | al bl

(define (pos? n) (>= 0))
(define (neg? n) (not (pos? n)))

(define (mul-interval a b)
  (let ((al (lower-bound a))
        (ah (upper-bound a))
        (bl (lower-bound b))
        (bh (upper-bound b)))
    (cond ((and (pos? al) (pos? ah) (pos? bl) (pos? bh))
           (make-interval (* al bl) (* ah bh)))
          ((and (pos? al) (pos? ah) (neg? bl) (pos? bh))
           (make-interval (* ah bl) (* ah bh)))
          ((and (pos? al) (pos? ah) (neg? bl) (neg? bh))
           (make-interval (* ah bl) (* al bh)))
          ((and (neg? al) (pos? ah) (pos? bl) (pos? bh))
           (make-interval (* al bh) (* ah bh)))
          ((and (neg? al) (pos? ah) (neg? bl) (pos? bh))
           (make-interval (* al bh) (* ah bh)))
          ((and (neg? al) (neg? ah) (pos? bl) (pos? bh))
           (make-interval (* al bl) (* al bl)))
          ((and (neg? al) (neg? ah) (neg? bl) (pos? bh))
           (make-interval (* al bl) (* al bl)))
          ((and (neg? al) (neg? ah) (neg? bl) (neg? bh))
           (make-interval (* ah bh) (* al bl)))
          ((and (neg? al) (pos? ah) (neg? bl) (pos? bh))
           (make-interval (* al bl) (* neg? bl)))))))

```

```

; our trouble case
(let ((p1 (* al bl))
      (p2 (* al bh))
      (p3 (* ah bl))
      (p4 (* ah bh)))
  (make-interval (min p1 p2 p3 p4)
                (max p1 p2 p3 p4))))))

```

What a mess. It might save on multiplications but it certainly didn't on programmer work, headaches, or readability.

ctz

Here is my code. It is a little simpler. But I still agree that Ben's advice is awful...

```

(define (mul-interval x y)
  (let ((x1 (lower x))
        (x2 (upper x))
        (y1 (lower y))
        (y2 (upper y)))
    (let ((x-neg (< x2 0))
          (x-pos (> x1 0))
          (y-neg (< y2 0))
          (y-pos (> y1 0)))
      (cond (x-neg (cond (y-neg (make-interval (* x2 y2) (* x1 y1)))
                           (y-pos (make-interval (* x1 y2) (* x2 y1)))
                           (else (make-interval (* x1 y2) (* x1 y1)))))
            (x-pos (cond (y-neg (make-interval (* x2 y1) (* x1 y2)))
                         (y-pos (make-interval (* x1 y1) (* x2 y2)))
                         (else (make-interval (* x2 y1) (* x2 y2)))))
            (else (cond (y-neg (make-interval (* x2 y1) (* x1 y1)))
                        (y-pos (make-interval (* x1 y2) (* x2 y2)))
                        (else (make-interval (min (* x1 y2) (* x2 y1))
                                             (max (* x1 y1) (* x2 y2)))))))))))

```

Hammer

Here's a slightly cleaned up solution of what rjk had. You'll notice it has 7 cond statements because the solution for  $-x_1, -x_2, +y_1, +y_2$ , is equivalent to  $-x_1, -x_2, -y_1, +y_2$ .

```

(define (pos? x) (< 0 x))
(define (neg? x) (> 0 x))

(define (mul-interval x y)
  (let ((x1 (lower-bound x))
        (x2 (upper-bound x))
        (y1 (lower-bound y))
        (y2 (upper-bound y)))
    (cond ((and (pos? x1) (pos? y1))
           (make-interval (* x1 y1) (* x2 y2)))
          ((and (pos? x2) (pos? y1))
           (make-interval (* x1 y2) (* x2 y1)))
          ((and (pos? x1) (pos? y2))
           (make-interval (* x2 y1) (* x2 y2)))
          ((and (neg? x2) (pos? y2))
           (make-interval (* x1 y2) (* x2 y1)))
          ((and (pos? x2) (neg? y2))
           (make-interval (* x1 y2) (* x2 y1)))
          ((and (neg? x2) (neg? y2))
           (make-interval (* x1 y1) (* x2 y2)))
          ((and (pos? x2) (pos? y2))
           (let ((i1 (* x1 y1))
                 (i2 (* x1 y2))
                 (i3 (* x2 y1))
                 (i4 (* x2 y2)))
             (make-interval (min i1 i2 i3 i4)
                           (max i1 i2 i3 i4)))))))

```

Qowl0

Here's an attempt of a more readable version by using an auxiliary procedure to test for the signs:

```

(define (mul-interval x y)
  (let ((+? positive?)
        (-? negative?))
    (lx (lower-bound x))
    (ly (lower-bound y))
    (ux (upper-bound x))
    (uy (upper-bound y)))
    (let ((i1 (* lx ly))
          (i2 (* lx uy))
          (i3 (* ux ly))
          (i4 (* ux uy)))
      (make-interval (min i1 i2 i3 i4)
                    (max i1 i2 i3 i4))))))

```

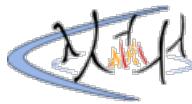
```

(uy (upper-bound y)))
(cond ((test-signs +? +? +? +? x y)
  (make-interval (* lx ly) (* ux uy)))
  ((test-signs +? +? -? +? x y)
  (make-interval (* ux ly) (* ux uy) ))
  ((test-signs -? +? +? +? x y)
  (make-interval (* lx uy) (* ux uy) ))
  ((test-signs -? -? +? +? x y)
  (make-interval (* lx uy) (* ux ly) ))
  ((test-signs +? +? -? -? x y)
  (make-interval (* ux ly) (* lx uy) ))
  ((test-signs -? +? -? -? x y)
  (make-interval (* ux ly) (* lx ly) ))
  ((test-signs -? -? -? +? x y)
  (make-interval (* lx uy) (* lx ly) ))
  ((test-signs -? -? -? -? x y)
  (make-interval (* ux uy) (* lx ly) ))
  ((test-signs -? +? -? +? x y)
  (let ((p1 (* lx ly))
    (p2 (* lx uy))
    (p3 (* ux ly))
    (p4 (* ux uy)))
  (make-interval (min p1 p2 p3 p4)
    (max p1 p2 p3 p4)))))

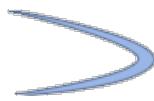
(define (test-signs lx-test ux-test ly-test uy-test x y)
  (and (lx-test (lower-bound x))
    (ux-test (upper-bound x))
    (ly-test (lower-bound y))
    (uy-test (upper-bound y))))
```

---

Last modified : 2021-11-17 17:20:34  
**WiLiKi 0.5-tekili-7** running on **Gauche 0.9**



# sicp-ex-2.12



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.11) | Index | Next exercise (2.13) >>

jz

There are a few ways to do this problem, e.g.:

1. totally redefine the internal structure of an interval, such that you cons the center and the percent
2. just define the constructor in terms of the width and percent, and add center and percent selectors

```
; ; ex 2.12

(define (make-interval a b) (cons a b))
(define (upper-bound interval) (max (car interval) (cdr interval)))
(define (lower-bound interval) (min (car interval) (cdr interval)))
(define (center i) (/ (+ (upper-bound i) (lower-bound i)) 2))

; ; Percent is between 0 and 100.0
(define (make-interval-center-percent c pct)
  (let ((width (* c (/ pct 100.0))))
    (make-interval (- c width) (+ c width)))

(define (percent-tolerance i)
  (let ((center (/ (+ (upper-bound i) (lower-bound i)) 2.0))
        (width (/ (- (upper-bound i) (lower-bound i)) 2.0)))
    (* (/ width center) 100)))

; ; A quick check:

(define i (make-interval-center-percent 10 50))
(lower-bound i)
(upper-bound i)
(center i)
(percent-tolerance i)

; The above returns
;Value: i
;Value: 5.
;Value: 15.
;Value: 10.
;Value: 50.
```

danylcraft

Do you think we need to abs the width as well? (this uses proportion instead of percent)

```
; e.g. for (make-center-percent -10 0.05)

(define (make-center-percent c p)
  (make-interval (- c (abs (* c p))) (+ c (abs (* c p)))))
```

qz

I think make-center-width, center, width can be used here to make it concise.

```
(define (make-center-percent c p)
  (make-center-width c (* c p)))
(define (percent i)
  (/ (width i) (center i)))
```

```
;We are talking about percentages rather than fractions. Therefore
```

```
(define (make-center-percent c p)
  (make-interval (- c (* c (/ p 100))))
    (+ c (* c (/ p 100)))))

;Also, for the percentage selector
(define (percent i)
  (* 100 (/ (width i) (center i))))
```

dxdm

Two points about calculating the percentage with

```
(/ (width i) (center i))
```

- Because of division by 0, the percentage is undefined if center happens to be 0. However, given a center of 0, make-center-percent produces a 0-width interval. If this is a valid result, the percent selector should also produce a valid result given a center and width of 0. (It's fine for other widths to be undefined, though.)
- The ratio of width and center can be found without calculating width and center:

```
width / center
= (2 * width) / (2 * center)
= (upper-bound - lower-bound) / (upper-bound + lower-bound)
```

vi

As dxdm mentions, if we have a center of zero, percentage in terms of a center which is zero is not well defined. Logically it is always zero, so I've defined a solution which lets the percentage be zero where the center is zero. Also, I don't use `width` as a procedure since they mentioned we'd only get to keep `center` as it appears in the previous definition:

```
(define (make-center-percent c p)
  (let ((width (abs (* (/ p 100) c))))
    (make-interval (- c width) (+ c width)))

(define (percent i)
  (if (= (center i) 0)
      0
      (* 100 (abs (/ (- (upper-bound i) (center i)) (center i))))))
```

# sicp-ex-2.13

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.12) | Index | Next exercise (2.14, 2.15, 2.16) >>

jz

No scheme, just math. If Ca is center of a, and Ta is tolerance of a, a is the interval

```
a = [Ca*(1 - 0.5*Ta), Ca*(1 + 0.5*Ta)]
```

and b is

```
b = [Cb*(1 - 0.5*Tb), Cb*(1 + 0.5*Tb)]
```

If the endpoints are positive, a\*b has the endpoints (after simplifying):

```
a*b = [Ca*Cb*(1 - 0.5*(Ta + Tb) + 0.25*Ta*Tb),
        Ca*Cb*(1 + 0.5*(Ta + Tb) + 0.25*Ta*Tb)]
```

Ta\*Tb will be a wee number, so it can be ignored. So, it appears that for small tolerances, the tolerance of the product will be approximately the sum of the component tolerances.

A quick check:

```
(define (make-interval a b) (cons a b))
(define (upper-bound interval) (max (car interval) (cdr interval)))
(define (lower-bound interval) (min (car interval) (cdr interval)))
(define (center i) (/ (+ (upper-bound i) (lower-bound i)) 2))

;; Percent is between 0 and 100.0
(define (make-interval-center-percent c pct)
  (let ((width (* c (/ pct 100.0))))
    (make-interval (- c width) (+ c width)))

(define (percent-tolerance i)
  (let ((center (/ (+ (upper-bound i) (lower-bound i)) 2.0))
        (width (/ (- (upper-bound i) (lower-bound i)) 2.0)))
    (* (/ width center) 100)))

(define (mul-interval x y)
  (let ((p1 (* (lower-bound x) (lower-bound y)))
        (p2 (* (lower-bound x) (upper-bound y)))
        (p3 (* (upper-bound x) (lower-bound y)))
        (p4 (* (upper-bound x) (upper-bound y))))
    (make-interval (min p1 p2 p3 p4)
                  (max p1 p2 p3 p4)))

(define i (make-interval-center-percent 10 0.5))
(define j (make-interval-center-percent 10 0.4))
(percent-tolerance (mul-interval i j))

;; Gives 0.89998, pretty close to (0.5 + 0.4).
```

sTeven

My method:

```
percent-interval :=> PI old-interval :=> OI
```

```
PI(a, p1) -> OI(a-a*p1, a+a*p1)
PI(b, p2) -> OI(b-b*p2, b+b*p2)
```

```
PI(c, p) = PI(a, p1)*PI(b, p2)
          :=> OI(a-a*p1, a+a*p1) * OI(b-b*p2, b+b*p2)
          = OI((a-a*p1)*(b-b*p2), (a+a*p1)*(b+b*p2))
          = OI(cL, cU)
c = (cL + cU)/2
  = ((a-a*p1)*(b-b*p2) + (a+a*p1)*(b+b*p2))/2
  = a*b*(1+p1*p2)
```

```
p = (c-cL)/c
= (a*b*(1+p1*p2) - (a-a*p1)*(b-b*p2)) / a*b*(1+p1*p2)
= a*b*(p1 + p2) / a*b*(1+p1*p2)
= (p1+p2) / (1+p1*p2)
```

```
c = a*b*(1+p1*p2);
p = (p1+p2) / (1+p1*p2);
```

**Example:**

```
PI(10, 0.005)*PI(10, 0.004)
= PI(10*10*(1+0.004*0.005), (0.004+0.005)/(1+0.004*0.005))
= PI(100.002, 0.00899982);
```

---

Last modified : 2015-09-08 11:23:34  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**

# sicp-ex-2.14-2.15-2.16

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.13) | Index | Next exercise (2.17) >>

Tengs

## Mathematical Explanation:

We can see it as a problem of finding extrema of n-variate functions. for R1, R2, R<sub>\_</sub>, ... these are variables, The interval of R1, R2, R<sub>\_</sub>, ... are the definition fields. The operations on variables are the function. And we want to find the maximum and minimum of the function in the definition fields.

For the general questions, It's very clear that for most functions, their extrema won't be found in the border, so we can't simple get the result by operating on lower and upper bound.

For the specific question there, the problem is that the functions Lem wrote assume the variables are independent. If the variables are dependent, the result is wrong.

For example, R1 = [1, 2]; R2 = R1 + 1=[2, 3], then R1 and R2 are dependent, so R1 - R2 = 1, and (sub-interval R1 R2) = [0, 2] is wrong.

Here, for par1, R1\*R2 and R1+R2 are dependent, so (div-interval R1\*R2 R1+R2) got wrong answer.

But, for pa2, [1,1] and R1; [1,1] and R2; 1/R1 and 1/R2; [1,1] and 1/R1 + 1/R2; they are all independent, so the result is right.

If we want let the result be correct, either we change the code fundamentally, or we need to ensure for each operation, their operands should be independent.

jz

All 3 problems point to the difficulty of "identity" when dealing with intervals. Suppose we have two numbers A and B which are contained in intervals:

A = [2, 8]  
B = [2, 8]

A could be any number, such as 3.782, and B could be 5.42, but we just don't know.

Now, A divided by itself must be 1.0 (assuming A isn't 0), but of A/B (the same applies to subtraction) we can only say that it's somewhere in the interval

[0.25, 4]

Unfortunately, our interval package doesn't say anything about identity, so if we calculated A/A, we would also get

[0.25, 4]

So, any time we do algebraic manipulation of an equation involving intervals, we need to be careful any time we introduce the same interval (e.g. through fraction reduction), since our interval package re-introduces the uncertainty, even if it shouldn't.

So:

2.14. Lem just demonstrates the above.

2.15. Eva is right, since the error isn't reintroduced into the result in par2 as it is in par1.

2.16. A fiendish question. They say it's "very difficult" as if it's doable. I'm not falling for that. Essentially, I believe we'd have to introduce some concept of "identity", and then have the program be clever enough to reduce equations. Also, when supplying arguments to any equation, we'd need to indicate identity somehow, since [2, 8] isn't necessarily the same as [2, 8] ... unless it is. Capiche?

shyam

A better explanation and some pointers to the interesting world of interval arithmetic  
[http://wiki.drewhess.com/wiki/SICP\\_exercise\\_2.16](http://wiki.drewhess.com/wiki/SICP_exercise_2.16)

voom4000

We have to leave the realm of interval arithmetic as soon as we start to talk about functions that have extrema somewhere inside intervals, or about functions like  $(A+B)/(A+C)$ , which in principle cannot be reduced to the form with one occurrence of each variable. This is the meaning of dependency problem. By using interval arithmetic we will always get wrong results for such functions. We cannot find correct answers using interval arithmetic, but we can find them by using analytical or numerical methods, e.g. Monte Carlo method. It's the problem of finding global minimum of a function of several variables. I saw the proposal to use MC in Weiqun Zhang's blog (in fact it is a standard method of finding extrema for very complex functions). MC works even when all analytical methods fail. Dependency problem does not exist for these methods. For example, when you calculate  $A/A$  with Monte Carlo method, you just submit the same random value inside the interval to ALL occurrences of A in the formula, e.g. for  $A/A$  the value of the function will be 1, no matter how many random values you generate. Do not work with intervals that span zero if you want to get correct result for that formula. Then you take the min/max of all function values (all of them were 1), and voila, final result is 1 with ZERO TOLERANCE or zero width. Well, you could cancel numerator and denominator from the very beginning, MC and analytical methods allow it.

HannibalZ

This is just my intuition: I think it is impossible, just like  $(A*B)*C \neq A*(B*C)$  if \* represents cross product.

pt

The problem about these algebraically equivalent ways of writing a formula is, that there is no inverse element (and no identity) in our interval multiplication, e.g.:

```
(define x (make-interval 1 10))
(mul-interval x (div-interval (make-interval 1 1) x))
=> (0.1 . 10.0)
```

The interval division is defined as if we had an inverse element. The algebraic transformations by Lem suppose that there was an inverse element in our interval multiplication.

vi

<https://stackoverflow.com/a/14131196/1449443> gives some good pointers to further study. I think it's difficult at my level of math knowledge to say that this problem is impossible to solve. At the same time, I don't want to go down the multi-variable diffeq rabbit hole far enough to solve it at the moment.

Tree3

I think this problem depends on the functional relationship between the two variables. If the variables in the two intervals are independent of each other, then this problem will not occur. So is it possible to consider introducing the relationship between the two variables into the interval operator in some form?

cat

Well guess my answer is no, this is not possible. I think the basic problem here is with identity interval and reverse operation. Suppose A and C are two intervals, then  $(A * C / C)$  equals A algebraically but not interval arithmetically. This is the same for  $(A + C - C)$ . Let C be interval of  $(lc, uc)$ , and suppose I have a new implementation of interval arithmetic. When I do subtraction in the case of  $(A + C - C)$ , I have to subtract lower-bound by  $lc$  (not  $uc$ ) and subtract upper-bound by  $uc$  (not  $lc$ ). See the problem is our simple implementation doesn't remember things or record what has been added before, so we really don't know if we should subtract by  $uc$  or  $lc$ . So, eh not possible.

kietsbe

I tried to create inverse object for intervals. It looks like illegal move resembling trick with imaginary unit (I'm not really worldly-wise in math however) this solution works great for me:

ex.2.16

```
(define (make-interval a b)
  (cons a b))

(define (lower-bound i)
  (car i))

(define (upper-bound i)
  (cdr i))

(define (width i)
  (/ (- (upper-bound i) (lower-bound i)) 2))

(define (center i)
  (/ (+ (lower-bound i) (upper-bound i)) 2))
```

```

(define (make-center-percent c p)
  (make-interval (- c (* c p)) (+ c (* c p)))))

(define (percent i)
  (/ (width i) (center i)))

(define (add-interval x y)
  (make-interval (+ (lower-bound x) (lower-bound y))
                (+ (upper-bound x) (upper-bound y)))))

(define (max a b)
  (if (> a b)
      a
      b))

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

(define (mul-interval x y)
  (if (and (>= (upper-bound x) (lower-bound x)) (>= (upper-bound y) (lower-bound y)))
      (let ((p1 (* (lower-bound x) (lower-bound y))))
        (p2 (* (lower-bound x) (upper-bound y)))
        (p3 (* (upper-bound x) (lower-bound y)))
        (p4 (* (upper-bound x) (upper-bound y))))
        (make-interval (min (min p1 p2) (min p3 p4))
                      (max (max p1 p2) (max p3 p4))))
        (make-interval (min (* (lower-bound x) (lower-bound y))
                           (* (upper-bound x) (upper-bound y)))
                          (max (* (lower-bound x) (lower-bound y))
                               (* (upper-bound x) (upper-bound y)))))))
      (lambda (i) (cons (/ 1 (lower-bound i)) (/ 1 (upper-bound i)))))

(define (div-interval x y)
  (if (<= (* (upper-bound y) (lower-bound y)) 0)
      (display "error")
      (mul-interval
        x
        (inverse-obj y)))))

(define (par1 r1 r2)
  (div-interval (mul-interval r1 r2)
                (add-interval r1 r2)))

(define (par2 r1 r2)
  (let ((one (make-interval 1 1)))
    (div-interval
      one (add-interval (div-interval one r1)
                        (div-interval one r2)))))

(define r1 (make-center-percent 500 0.88))
(define r2 (make-center-percent 3000 0.42))

(par1 r1 r2)
(par2 r1 r2)

```

hope it will be helpful

chemPolonium

This is a great work which use the trick on the order of "lower" bound and "upper" bound. However, when it comes to division on two different numbers, which will indeed make a interval, this method will put the wrong interval:

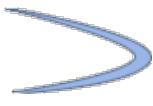
```

(define r1 (make-center-percent 500 0.88))
(define r2 (make-center-percent 500 0.88))
; this will produce a zero width interval
(div-interval r1 r2)

```



# sicp-ex-2.17



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercises (2.14-2.15-2.16) | Index | Next exercise (2.18) >>

```
;; ex-2.17

(define (last-pair items)
  (let ((rest (cdr items)))
    (if (null? rest)
        items
        (last-pair rest)))

;; Testing
(last-pair (list 23 72 149 34)) ; (34)
```

jz  
;; ex 2.17  
(define (last-pair items)
 (let ((rest (cdr items)))
 (if (null? rest)
 items
 (last-pair rest)))

;; Test:
(last-pair (list 1 2 3 4))

asb  
solution which doesn't fail for '()'  
(define (last-pair items)
 (define (iter items result)
 (if (null? items)
 result
 (iter (cdr items) items)))
 (iter items items))

otakutyrant A lot of people here concern such a corner case that the list is empty, but the exercise dictates conversely:

> Define a procedure last-pair that returns the list that contains only the last element of a given (\*\*nonempty\*\*) list:

Speaking of it, I cannot imagine what the last element is while the list is empty exactly. It is contradictory that an empty list contains an empty list as its latest element inside.

EcsCodes A simple but ingenious code

```
(define (lastPair L)
  (if (= (length L) 1) (car L)
      (lastPair (cdr L))))
```

kaiix same as EcsCodes, but passes test for empty list

```
(define (last-pair items)
  (if (< (length items) 2) items
      (last-pair (cdr items))))
```

FPaul  
;; ex 2.17  
(define (lastPair L)

```

(define (ref lst x)
  (if (= x 0) (car lst)
      (ref (cdr lst) (- x 1)))
  (if (= (length L) 1) (car L)
      (cons (ref L (- (length_ L) 2))
            (cons (ref L (- (length_ L) 1)) '())))
      )
  )
)

```

atrika

I think using `(length L)` repeatedly is not a good idea (unless your length is  $O(1)$ , but the implementation in the book is  $O(n)$ ) because it makes `lastPair` do  $n!$  length-iter on the list, its rest, the rest of the rest... where  $n$  is the size of the items list.

AMS

The answer was meant to return the last item of a list as a list. So you must modify the return value to return a list.

```

(define (last-pair l)
  (cond ((null? l) l)
        ((null? (cdr l)) (list (car l)))
        (else (last-pair (cdr l)))))

;; test
(last-pair (list 23 72 149 34)) ;;= (34)
(last-pair '()) ;;= ()

```

shubhro

Without error checking though.

```

(define (last-pair s)
  (if (null? (cdr (cdr s)))
      (cdr s)
      (last-pair (cdr s))))

```

Daniel-Amarie1

Regarding the first solution. We need to return a list, not a particular element.  $1 \neq '(1)$

```

;; returns the list that contains the last element
;; of a given non-empty list

(define (last-pair L)
  (if (null? (cdr L))
      L
      (last-pair (cdr L))))

(last-pair (list 1)) ;; '(1)
(last-pair (list 1 2)) ;; '(2)
(last-pair (list 1 2 3)) ;; '(3)
(last-pair (list 1 2 3 4)) ;; '(4)

```

Anonymous coward

Does the solution have to be recursive?

```

(define (last-pair inlist)
  (list (list-ref inlist (- (length inlist) 1))))

```

anon

racket with contracts

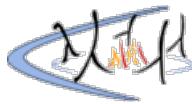
```

(define/contract (last-pair items)
  (->i ([items list?])
    #:pre (items) (not (null? items))
    [_ list?])
  (if (null? (cdr items))

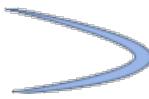
```

```
    items
  (last-pair (cdr items))))  
  
(check-equal? (last-pair (list 23 72 149 34)) '(34))  
(last-pair '()) ; contract violation
```

```
bmm          ;; ex 2.17  
  
(define nil '())  
  
;; procedure for length of list  
  
(define length (lambda (list)
  (if (null? list) 0
      (+ 1 (length (cdr list)))))  
  
;; list-ref
(define list-ref (lambda (list n)
  (if (= n 0) (car list)
      (list-ref (cdr list) (- n 1)))))  
  
;; last-pair procedure
(define last-pair (lambda (list)
  (if (null? list) nil
      (let ((s (- (length list) 1)))
        (cons (list-ref list s) nil)))))  
  
;; Testing
(last-pair (list 1 2 3 4 5 6 7 8 9)) ;; '(9)
```



# sicp-ex-2.18



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.17) | Index | Next exercise (2.19) >>

jz

I couldn't figure out a way to do this problem without using an intermediary variable to store the reversed list as it is built. If there is such a way, post it.

```
; ex 2.18, reverse.  
  
;; Note: nil isn't defined in my environment, using footnote from page  
;; 101.  
(define nil '())  
  
(define (reverse items)  
  (define (iter items result)  
    (if (null? items)  
        result  
        (iter (cdr items) (cons (car items) result))))  
  
  (iter items nil))  
  
;; Usage  
(reverse (list 1 2 3 4))
```

vlprans

This is a solution that doesn't use intermediary variable, former should be more efficient though.

```
(define nil '())  
  
(define (reverse items)  
  (if (null? (cdr items))  
      items  
      (append (reverse (cdr items))  
              (cons (car items) nil))))  
  
(reverse (list 1 2 3 4))
```

You can get rid of the nil definition by replacing cons with list:

```
guile>  
(define (reverse items)  
  (if (null? (cdr items))  
      items  
      (append (reverse (cdr items))  
              (list (car items)))))  
guile> (reverse (list 1 2 3 4))  
(4 3 2 1)  
guile>
```

You can also get rid of the boundary-condition failure by eliminating the first cdr:

```
guile> (reverse '())  
  
Backtrace:  
In standard input:  
 9: 0* [reverse {()}]  
 2: 1  (if (null? (cdr items)) items ...)  
 2: 2* [null? ...]  
 2: 3*  [cdr {()}]  
  
standard input:2:14: In procedure cdr in expression (cdr items):  
standard input:2:14: Wrong type (expecting pair): ()
```

```

ABORT: (wrong-type-arg)
guile>
(define (reverse items)
  (if (null? items)
      items
      (append (reverse (cdr items))
              (list (car items)))))

guile> (reverse '())
()
guile> (reverse (list 1 2 3 4))
(4 3 2 1)
guile>

```

bmm ;; ex 2.18

```

;; nil
(define nil '())

;; returns length of list
(define length (lambda (list)
  (if (null? list) 0
      (+ 1 (length (cdr list))))))

;; returns the n-th element
(define list-ref (lambda (list n)
  (if (= n 0) (car list)
      (list-ref (cdr list) (- n 1)))))

;; reverse list procedure
(define (reverse list)
  (define (do-reverse items size)
    (if (< size 0) nil
        (cons (list-ref items size) (do-reverse items (- size 1)))))

  (let ((len (- (length list) 1)))
    (do-reverse list len)))

;; Test

(reverse (list 1 2 3 4 5)) ;; '(5 4 3 2 1)

```

karthikk Yes, the boundary condition (when the input parameter is the null list) should work. So here are two simple solutions: the first generates an iterative process and the second a recursive process (note since cons treats its first formal parameter as an atom, we have to use append in the recursive definition). The iterative solution is mostly the same as the fp's but the recursive definition eliminates testing the base step on the cdr of the list...

```

;iterative solution
(define (i-reverse l)
  (define (it-rev lat ans)
    (if (null? lat)
        ans
        (it-rev (cdr lat) (cons (car lat) ans))))
  (it-rev l '()))

;recursive solution
(define (r-reverse lat)
  (if (null? lat)
      '()
      (append (r-reverse (cdr lat)) (list (car lat)))))


```

dgski Solution using collector paradigm (form of iterative process).

```

(define (reverse l)
  (define (helper l col)
    (if (null? (cdr l))
        (col (car l))
        (helper (cdr l) (lambda (x)
                           (cons x (col (car l)))))))
  (helper l (lambda (x) (cons x '()))))

```

gwj a iterative solution of reverse,also fit when it is a empty list

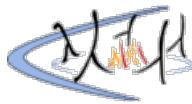
```
(define (reverse items)
  (define (iter list result)
    (if (null? list)
        result
        (iter (cdr list) (cons (car list) result))))
  (if (null? items)
      (list )
      (iter (cdr items) (cons (car items) nil))))
```

---

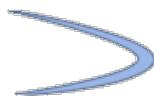
[\*\*<< Previous exercise \(2.17\)\*\*](#) | [\*\*Index\*\*](#) | [\*\*Next exercise \(2.19\) >>\*\*](#)

---

Last modified : 2021-08-14 15:01:30  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-2.19



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.18) | Index | Next exercise (2.20) >>

jz  
;; Counting change.

```
; Helper methods, could define them in cc too:  
(define (first-denomination denominations) (car denominations))  
(define (except-first-denom denominations) (cdr denominations))  
(define (no-more? denominations) (null? denominations))  
  
(define (cc amount denominations)  
  
  
  
    ; 292  
  
    ; 104561 ... wow.
```

atrika

this also works

```
(define no-more? null?)  
(define except-first-denomination cdr)  
(define first-denomination car)
```

Rptx

For the last part of the exercise. The order of the coins does not affect the result. Because the procedure computes all possible combinations. But it does affect the speed of the computation. If you start with the lower valued coins, it'll take much longer.

```
(define (timed-cc amount coin-values start-time)  
  (cc amount coin-values)  
  (- (runtime) start-time))  
  
            ((timed-cc 100 us-coins (runtime)))  
            (display "Reverse takes longer"))  
            (display "Reverse does not take longer")) ;As expected, reverse takes longer
```

Ely

@Rptx: It is slower because of the recursive calls calling **reverse** everytime. If I put a list in reversed order as argument I do not notice any speed penalty.

## Using other methods to achieve the same function

```

;; functional programming
(define (count-change-1 amount denomination-list)
  (length (filter
            (lambda (data) (= amount (apply + (map * data denomination-list))))
            (accumulate
              (lambda (a-list b-list)
                (flatmap
                  (lambda (a) (map (lambda (b) (cons a b)) b-list))
                  a-list))
              (list '()))
            (map
              (lambda (c) (enumerate-interval 0 c))
              (map (lambda (d) (floor (/ amount d))) denomination-list)))))

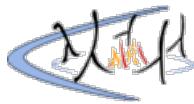
;; Using nested mappings

(define (nested-map mapping op keys-list)
  (define (rec keys-list args)
    (if (null? keys-list)
        (apply op (reverse args))
        (mapping
          (lambda (key)
            (rec (cdr keys-list) (cons key args)))
          (car keys-list))))
    (rec keys-list '())))

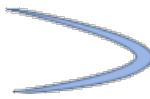
(define (count-change-2 amount denomination-list)
  (let ((count 0))
    (define (count-inc! . b)
      (if (= amount (apply + (map * b denomination-list)))
          (set! count (+ count 1))))
    (nested-map
      for-each
      count-inc!
      (map
        (lambda (c) (enumerate-interval 0 c))
        (map (lambda (d) (floor (/ amount d))) denomination-list)))
    count)))

```

[<< Previous exercise \(2.18\)](#) | [Index](#) | [Next exercise \(2.20\) >>](#)



# sicp-ex-2.20



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.19) | Index | Next exercise (2.21) >>

```
(define (same-parity first . rest)
  (define (same-parity-iter source dist remainder-val)
    (if (null? source)
        dist
        (same-parity-iter (cdr source)
                          (if (= (remainder (car source) 2) remainder-val)
                              (append dist (list (car source)))
                              dist)
                          remainder-val)))

  (same-parity-iter rest (list first) (remainder first 2))))
```

mueen

Using append as above is expensive, as it will iterate through all the elements of the list you build, at \_each\_ iteration.

It's better simply to build the list in reverse order, and then reverse the final list at the end

```
(define (same-parity first . rest)
  (let ((yes? (if (even? first)
                  even?
                  odd?)))
    (define (iter items result)
      (if (null? items)
          (reverse result)
          (iter (cdr items) (if (yes? (car items))
                               (cons (car items) result)
                               result))))
    (iter rest (list first))))
```

```
(define (sam-parity first . rest)
  (define (inter yes? lat)
    (cond
      ((null? lat) (quote()))
      ((yes? (car lat)) (cons (car lat) (inter yes? (cdr lat)))))
      (else
        (inter yes? (cdr lat)))))
  (if (odd? first)
      (inter odd? rest)
      (inter even? rest))))
```

jz

```
;; ex 2.20, dotted-tail notation.

(define (same-parity first . rest)
  (define (congruent-to-first-mod-2? a)
    (= (remainder a 2) (remainder first 2)))

  (define (select-same-parity items)
    (if (null? items)
        items
        (let ((curr (car items))
              (select-rest (select-same-parity (cdr items))))
          (if (congruent-to-first-mod-2? curr)
              (cons curr select-rest)
              select-rest)))

    (cons first (select-same-parity rest)))

  ;; an alternative implementation by andras:
  (define (same-parity a . l)
    (define (sp-builder result tail)
      (if (null? tail)
          result
          (if (even? (+ a (car tail))))
```

```

;;test for same parity
;;if the current beginning of the rest (car tail) is the same parity as "a", then
it is appended to the result, else the result is left untouched
(sp-builder (append result (list (car tail))) (cdr tail))
(sp-builder result (cdr tail))))
(sp-builder (list a) l))

;; Usage:
(same-parity 1 2 3 4 5 6 7)
;; (1 3 5 7)

(same-parity 2 3 4 5 6 7 8)
;; (2 4 6 8)

```

chris

This does it by passing the relevant test:

```

(define (same-parity . l)
(define (parity l test)
(if (null? l)
(list)
(if (test (car l))
(cons (car l)(parity (cdr l) test))
(parity (cdr l) test)))

(if (even? (car l))
(parity l even?)
(parity l odd?)))

```

pritesh

```

(define (same-parity x . y)
;; Finds the list of elements in y with same parity as x
(define (same-parity-rest y)
(cond
(
  (null? y) '()
)
(
  (parity-2 (car y) x)
  (cons (car y) (same-parity-rest (cdr y)) )
)
(
  else ;;(car y) has different parity than x..
  (same-parity-rest (cdr y))
)
)
)
;; Check if 2 integers have same parity
(define (parity-2 a b)
(= (remainder (+ a b) 2) 0) ;; is thier sum even ?
)
;; Combine the result
(cons x (same-parity-rest y))
)
```

erik

```

(define (same-parity n . lst)
(define same-parity? (if (even? n) even? odd?))
(define (iter lst acc)
(if (null? lst)
acc
(let ((first (car lst))
(rest (cdr lst)))
(iter rest
(if (same-parity? first)
(cons first acc)
acc))))))
(cons n (reverse (iter lst null))))
```

The footnote for this section shows a funny lambda:

```
(define f (lambda (x y . z) z))
(f 1 2 3 4 5)
=> (3 4 5)
```

Defining a lambda that takes a variable number of args is isn't done in the obvious way:

```
(define h (lambda (. w) w))
;Ill-formed dotted list: (. w) ... etc, lots of errors.
```

This has to be defined like this:

```
(define h (lambda w w))
(h 1 2 3 4)
=> (1 2 3 4)
```

**Shubhro**

```
(define (same-parity x . y)
  (define (parity list rem)
    (cond ((null? list) list)
          ((= rem (remainder (car list) 2))
           (cons (car list) (parity (cdr list) rem) ))
          (else
            (parity (cdr list) rem))))
    (if (even? x)
        (parity y 0)
        (parity y 1)))
```

Daniel-Amariei

### Recursive process

```
(define (same-parity . L)
  (define (filter x)
    (if (even? x)
        even?
        odd?))
  (define (construct f L)
    (cond ((null? L) '())
          ((f (car L)) (cons (car L)
                               (construct f (cdr L)))))
          (else (construct f (cdr L))))))
  (construct (filter (car L)) L))

(same-parity 2 3 4 5 6 7) ;; '(2 4 6)
```

wind2412

### My solution.

```
(define (same-parity x . w)
  (define (get-all w r)
    (if (null? w) `()
        (if (= (remainder (car w) 2) r) (cons (car w) (get-all (cdr w) r))
            (get-all (cdr w) r))))
  (cons x (get-all w (remainder x 2))))
```

It is possible to test for parity using just the sum of two terms (odd+odd=pair pair+pair=pair). Below Using the sum of first and each new term.

```
(define (same-parity first . l)
  (define (iter lista lfinal)
    (if (null? lista)
        lfinal
        (if (even? (+ first (car lista)))
            (iter (cdr lista) (append lfinal (list (car lista)))))
            (iter (cdr lista) lfinal))))
  (iter l (list first)))
```

acml

### My recursive solution.

```
(define (same-parity x . y)
```

```
(define (search-parity a r)
  (if (null? r)
      '()
      (if (= (remainder a 2) (remainder (car r) 2))
          (cons (car r) (search-parity a (cdr r)))
          (search-parity a (cdr r))))
  (cons x (search-parity x y))))
```

ly

```
(define (find-numbers items condition)
  (if (null? items)
      '()
      (list)
      (if (condition (car items))
          (cons (car items) (find-numbers (cdr items) condition))
          (find-numbers (cdr items) condition)))))

(define (same-parity first . items)
  (if (odd? first)
      (find-numbers items odd?)
      (find-numbers items even?)))
```

Here is my solution

```
(define (same-parity first . items)
  (define (iter items)
    (if (null? items)
        '()
        (if (even? (+ (car items) first))
            (cons (car items) (iter (cdr items)))
            (iter (cdr items))))
  (cons first (iter items))))
```

Evan

depaulagu

solution using filter

```
(define (same-parity . l)
  (define (first-parity)
    (if (even? (car l))
        even?
        odd?))
  (filter (first-parity) l))
```

stewoe

Storing the function to apply (even? or odd?) as predicate and applying it using filter

```
(define (same-parity fst . rest)
  (let ((pred (if (even? fst) even? odd?)))
    (filter pred (cons fst rest))))
```

thongpv87

My solution using recursion

```
(define (same-parity i . l)
  (define (same-parity-i x)
    (if (even? (+ i x)) #t #f))
  (cond ((null? l) i)
        ((same-parity-i (car l))
         (cons i (apply same-parity l)))
        (else (apply same-parity i (cdr l)))))
```

Marisa

This solution leverages abstraction, by introducing a `sub-list` procedure that returns a subset of the input that matches a predicate. `same-parity` is defined in terms of `sub-list`.

```
(define (sub-list predicate? x)
```

```

(define (sub-list-iter items result)
  (if (null? items) (reverse result)
      (let ((caritems (car items)))
        (cond
          ((predicate? caritems)
           (sub-list-iter (cdr items)
                         (cons caritems result)))
          (else
           (sub-list-iter (cdr items) result))))))
  (sub-list-iter x nil))

(define (same-parity x . A)
  (if (even? x)
      (sub-list even? A)
      (sub-list odd? A)))

```

2bdkid

Works but isn't too pretty

```

(define (same-parity x . y)
  (define (filter p items)
    (cond ((null? items) #nil)
          ((p (car items)) (cons (car items) (filter p (cdr items))))
          (else (filter p (cdr items)))))
  (let ((parity (remainder x 2)))
    (cons x (filter (lambda (x) (= parity (remainder x 2))) y))))

```

It's surprising how many of the above solutions use functions that have not been introduced in the book at this point. Here's my solution:

```

(define (same-parity . args)
  (define (same-parity-step parity lst)
    (cond ((null? lst) NIL)
          ((= parity (remainder (car lst) 2)) (cons (car lst) (same-parity-step
parity (cdr lst))))
          (else (same-parity-step parity (cdr lst)))))
    )
  (same-parity-step (remainder (car args) 2) args)
)

```

joshroybal

The way I did it.

```

(define (same-parity a . b)
  (define (iter b result)
    (cond ((null? b)
           (reverse result))
          ((or (and (even? a) (even? (car b)))
               (and (odd? a) (odd? (car b))))
              (iter (cdr b) (cons (car b) result)))
          (else
           (iter (cdr b) result))))
    (iter b (list a)))

```

Or non-iteratively.

```

(define (same-parity a . b)
  (define (aux b)
    (cond ((null? b)
           b)
          ((or (and (even? a) (even? (car b)))
               (and (odd? a) (odd? (car b))))
              (cons (car b) (aux (cdr b))))
          (else
           (aux (cdr b)))))
    (cons a (aux b)))

```

yc

```

(define (same-parity a . list)
  (define (reverse list)
    (define (iter source result)
      (if (null? source)
          result

```

```
(iter (cdr source) (cons (car source) result)))
(iter list()))
(define (construct test source result)
  (if (null? source)
      (reverse result)
      (if (test (car source))
          (construct test (cdr source) (cons (car source) result))
          (construct test (cdr source) result))))
(cons a (construct
  (if (even? a) even? odd?)
  list
  ()))))
```

---

Last modified : 2022-01-03 14:55:26  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**

# sicp-ex-2.21

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.20) | Index | Next exercise (2.22) >>

jz

```
(define (square-list items)
  (if (null? items)
      items
      (cons (square (car items)) (square-list (cdr items)))))

(square-list (list 1 2 3 4))

(define (sq2 items)
  (map (lambda (x) (square x)) items))

(sq2 (list 1 2 3 4))
```

```
(define (sq3 items)
  (map square items))

(sq3 (list 1 2 3 4))
```

deaulagu

```
(define nil '())

(define (square-list items)
  (if (null? items)
      nil
      (cons (square (car items))
            (square-list (cdr items)))))

(define (square-list-m items)
  (map square items))

(square-list (list 1 2 3 4 5 6 7 8 9 10))
(square-list-m (list 1 2 3 4 5 6 7 8 9 10))
```

# sicp-ex-2.22

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.21) | Index | Next exercise (2.23) >>

jz

```
(define nil '())

(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons (square (car things)) answer))))
  (iter items nil))

(square-list (list 1 2 3 4))

;; The above doesn't work because it conses the last item from the
;; front of the list to the answer, then gets the next item from the
;; front, etc.

(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons answer (square (car things)))))))
  (iter items nil))

(square-list (list 1 2 3 4))

;; This new-and-not-improved version conses the answer to the squared
;; value, but the answer is a list, so you'll end up with (list (list
;; ...) lastest-square).
```

shyam

Saying in a different way, the bug on the second version can also be attributed to the property of cons as shown below.

```
1 ]=> (cons 1 2)
;Value 11: (1 . 2)

1 ]=> (cons 1 (list 2 3))
;Value 12: (1 2 3)

1 ]=> (cons (list 2 3) 1)
;Value 13: ((2 3) . 1)
```

somarl

An implementation evolving an iterative process works.

```
(define (square-list items)
  (define (iter l pick)
    (define r (square (car l)))
    (if (null? (cdr l))
        (pick (list r))
        (iter (cdr l) (lambda (x) (pick (cons r x))))))
  (iter items (lambda (x) x)))
```

roy-tobin

An innovative solution. Please consider, from a tyro, the following polish which doesn't fail on the empty list.

```
(define (square-list5 items)
```

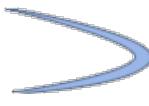
```
(define (iter l pick)
  (if (null? l)
      (pick l)
      (let ((r (square (car l))))
        (iter (cdr l) (lambda (x) (pick (cons r x)))))))
  (iter items (lambda (x) x)))
```

Tengs

the problem here is that Louis uses iteration and now we can only use `cdr` to get the rest of the list, so we must process the list from the tail by iteration, and since the `answer` is originally `nil`, we can't just change the order to `(cons answer (square (car things)))` because when the program executed to the innermost iteration, the first element will be `nil`, so the result will be like (((() . 1) . 4) . 9)



# sicp-ex-2.23



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.22) | Index | Next exercise (2.24) >>

madschemer

why complicate use "and" and output some dummy return value

```
(define (for-Each proc items)
  (if (null? items)
      #t
      (and (proc (car items)) (for-Each proc (cdr items))))))
```

simv

I think the above solution by madschemer has a bug. If the procedure you pass in returns #t it won't recurse down the list, because the call to "and" will be short-circuited.

```
(define (for-each proc items)
  (let ((items-cdr (cdr items)))
    (proc (car items))
    (if (not (null? items-cdr))
        (for-each proc items-cdr)
        true)))
```

sritchie

```
; I didn't find that any of the other implementations here
; supported the null list as input; here's my fix.
; for-each

(define (for-each proc list)
  (cond
    ((null? list) #t)
    (else (proc (car list))
          (for-each proc (cdr list))))))
```

jz

```
; for-each

; The below didn't work ... basically, I needed some kind of block
; structure, since if has the form (if (test) true-branch
; false-branch). I needed to have true-branch execute the proc, then
; call the next iteration of for-each, and the only way I knew how to
; do that was with brackets ... but of course that doesn't work, as
; the interpreter tries to apply the result of the first proc call as
; a function to the rest.
(define (for-each proc items)
  (if (not (null? items))
      ((proc (car items))
       (for-each proc (cdr items)))))

(for-each (lambda (x) (newline) (display x)) (list 1 2 3 4))

; This one works.
; Moral: cond is better for multi-line branches.
(define (for-each proc items)
  (cond ((not (null? items))
         (proc (car items))
         (for-each proc (cdr items)))))
```

tyg

```
;;; First, I'm sorry for using common lisp.  
;;; This is an easy but funny problem. At a first glance, I think I need some  
;;; block structure such as cond, progn, etc., in fact, you can use function  
;;; argument eval rule to avoid using them at all. I think this solution has  
;;; more 'functional style'.
```

```
(defun for-each (f items)  
  (labels ((iter (action lst)  
             (if (null lst)  
                 action  
                 (iter (funcall f (car lst)) (cdr lst))))  
          (iter nil items)))
```

denziloe

Like tyg's answer but using Scheme.

```
; Simply iterate with a parameter solely for evaluating the procedure with the current  
argument.  
; In accordance with the function evaluation rule, this parameter will be evaluated each  
time.  
; The parameter is not used in the function body -- it is 'discarded'.  
; The initial iteration passes #t to this parameter, which has no effect.  
  
(define (for-each procedure items)  
  (define (iter items evaluate)  
    (if (null? items)  
        #t  
        (iter (cdr items) (procedure (car items)))))  
  (iter items #t))
```

amasad

```
; :)  
  
(define (for-each proc items)  
  (map proc items)  
  #t)
```

otakutyrant

Same idea but an explicit distort function.

```
(define  
  (for-each procedure list_)  
  (define (distort x) #t)  
  (distort (map procedure list_))  
)  
(for-each (lambda (x) (newline) (display x)) (list 57 321 88))
```

AMS

The above solution by amasad is incorrect as MAP is different to FOR-EACH. Map will create a LIST of the results whereas we don't want the output of For-each to create a list, just to apply the specified procedure to each of the list elements.

Gera

The solution by amasad is perfectly fine. It returns true, it doesn't matter if internally it "creates a list" or whatever. In fact, I'd say this'd be fine too:

```
(define for-each map)
```

as the exercise states that the result can be arbitrary, so it can arbitrarily be the list of items in the list applied to the given procedure.

I think this solution is incorrect not because of AMS's reason, but rather because the

THIS SOLUTION IS INCORRECT NOT BECAUSE OF AMIGO'S REASON, BUT RATHER BECAUSE THE implementation of MAP can vary, and it isn't guaranteed that the procedure that is given to MAP will be applied left-to-right (see **this** in the R5RS specification: "The dynamic order in which proc is applied to the elements of the lists is unspecified").

wbooze

```
;;; i did the same with common-lisp  
;;; it pretty much matches the first scheme form on this page tho!  
  
(defun for-each (proc items)  
  (if (null items)  
      nil  
      (apply proc (car items) nil))  
  (if (not (null (cdr items)))  
      (for-each proc (cdr items))))
```

anonymous

```
; Another way to use if instead of cond.  
; We just wrap the multiple statements in a let.  
; From: http://wiki.drewhess.com/wiki/SICP\_exercise\_2.23  
  
(define (for-each proc items)  
  (if (not (null? items))  
      (let ()  
        (proc (car items))  
        (for-each proc (cdr items))))
```

anonymous you can use "begin" instead of "let ()"

```
(begin  
  (...)  
  (...))
```

coriolis

```
(define (for-each unary-proc seq)  
  (if (null? seq)  
      (newline)  
      (unary-proc (car seq)) (for-each unary-proc (cdr seq)) ) )
```

erik

This was my solution, kind of weird but it worked on the example in the book.

```
(define (for-each f items)  
  (define (iter f items . cur)  
    (if (null? items)  
        #t  
        (iter f  
              (cdr items)  
              (f (car items)))))  
  (iter f items))
```

Tom

May be not so concise as other solutions. But without let and other advanced features, very straightforward. The "block structure" mentioned by jz can be achieved by a function, as we have learnt from previous chapter (procedural abstraction).

```
(define (for-each proc items)
  (if (null? items)
      nil
      ((lambda (li) (proc (car li)) (for-each proc (cdr li))) items)))
```

And in the first answer, do not see why a return value true is necessary. The following one will not return or print a #t as the end.

```
(define (for-each-an1 proc items)
  (let ((items-cdr (cdr items)))
    (proc (car items))
    (if (not (null? items-cdr))
        (for-each-an1 proc items-cdr)
        )))
```

master

Not really sure if this is an elegant solution but it seems much cleaner than all the other solutions that have been posted. Using the print procedure given as an example, if the input is the empty list then it prints that, otherwise it prints every element of the list excluding the empty list. It works because it first checks whether items is the empty list before reaching ahead to the cdr of items. I don't see any other way to prevent passing a non-pair to cdr and still treat nil as the list terminator.

```
(define (for-each proc items)
  (cond ((null? items) (proc items))
        ((null? (cdr items)) (proc (car items)))
        (else (proc (car items)) (for-each proc (cdr items)))))
```

wilson

I use lambda compose two statement in else branch

```
(define (for-each proc list)
  (if (null? list)
      '()
      ((lambda ()
        (proc (car list))
        (for-each proc (cdr list))))))
```

justinm

This solution avoids advanced language features and illustrates a general problem solving approach: First, think of a precondition which makes the problem easy to solve (in this exercise, when the list is nonempty). Then solve the general case by trying to satisfy the precondition.

```
(define (for-each proc list)
; precondition: list is nonempty.
(define (for-each-nonempty list)
  (proc (car list))
  (if (null? (cdr list))
      '()
      (for-each-nonempty (cdr list))))
; general case: ensure precondition.
(if (null? list)
    '()
    (for-each-nonempty list)))
```

gustave

take advantage of the fact that each cond clause may be a sequence of expressions

```
(define (for-each fxn items)
  (cond ((null? items) #t)
        (else (fxn (car items)) (for-each fxn (cdr items))))))
```

Clean and no empty list returned

svelty

```
(define (for-each proc items)
  (cond ((null? items) nil)
        ((null? (cdr items)) (proc (car items)))
        (else (proc (car items)) (for-each proc (cdr items)))))
```

Last modified : 2022-08-21 23:24:14  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-2.24

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.23) | Index | Next exercise (2.25) >>

Tree:

```
(1 (2 (3 4)))  
  ^  
 / \  
1   (2 (3 4))  
 / \  
2   (3 4)  
 / \  
3   4
```

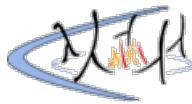
jz

```
(list 1 (list 2 (list 3 4)))
```

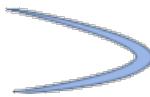
Printed result: (1 (2 (3 4)))

Box and pointer:

```
(1 (2 (3 4))) ((2 (3 4)))  
+---+---+ +---+---+  
| * | *--->| * | / |  
+---+---+ +---+---+  
| | |  
V V (2 (3 4)) ((3 4))  
+---+ +---+---+ +---+---+  
| 1 | | * | *--->| * | / |  
+---+ +---+---+ +---+---+  
| | |  
V V (3 4)  
+---+ +---+---+ +---+---+  
| 2 | | * | *--->| * | / |  
+---+ +---+---+ +---+---+  
| | |  
V V  
+---+ +---+  
| 3 | | 4 |  
+---+ +---+
```



# sicp-ex-2.25



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.24) | Index | Next exercise (2.26) >>

jz

2.25 seems pretty easy, but it gets rather hairy.

a.

```
(define items (list 1 3 (list 5 7) 9))  
(car (cdr (car (cdr (cdr items)))))  
;; => 7
```

The final car (on the left) is needed to pull the 7 from the (cons 7 nil) list.

To build the cons equivalent of the original:

Step 1: replace (list 5 7) with x:

```
(cons 1 (cons 3 (cons x (cons 9 nil))))
```

Step 2: replace x with cons equivalent:

```
(define nil '())  
(cons 1 (cons 3 (cons (cons 5 (cons 7 nil)) (cons 9 nil))))
```

b.

```
(car (car (list (list 7))))  
;; => 7
```

c.

```
(define a (list 1 (list 2 (list 3 (list 4 (list 5 (list 6 7)))))))  
;; a => (1 (2 (3 (4 (5 (6 7))))))
```

I thought the answer was just:

```
(car (cdr (cdr (cdr (cdr (cdr a)))))))
```

but no.

Building up from the inside out:

```
(6 7) is (list 6 7)
```

is by definition

```
(cons 6 (cons 7 nil))
```

```
(5 (6 7)) is (list 5 (list 6 7))
```

Replace (list 6 7) with x for a second to see this is just

```
(cons 5 (cons x nil))
```

Resubstitute the expanded expression for x:

```
(cons 5 (cons (cons 6 (cons 7 nil)) nil))  
----- z -----
```

So, any time we add another "wrapping list" we'll need to add the underlined portions, replacing z with the appropriate value.

Therefore,

```
(list 1 (list 2 (list 3 (list 4 (list 5 (list 6 7))))))
```

is equivalent to

```
(cons 1 (cons (cons 2 (cons (cons 3 (cons (cons 4 (cons (cons 5 (cons (cons 6 (cons 7  
nil)) nil)) nil)) nil)) nil)) nil)))
```

Which is pretty ugly (actually, I'll call this "ugly" in the explanation below).

Anyway, to retrieve the 7, we need to do the following:

```
(car (cdr (car (cadr (car (cadr (car (cadr (car (cadr (car (cadr (car (cadr a)))))))))))))))
```

Which is also frightful.

### Explanation:

The innermost cdr gives the section after the number 1 in ugly, or "(cons (cons 2 ...".

The next car gives "(cons 2 (cons ...)).

The next cdr gives "(cons (cons 3 ...".

The next car gives "(cons 3 ...)", etc.

Eventually, we get to (cons 7 nil), which is the final car.

Yuck.

time

tim to leave a list, you would car. to move up the list, you would cdr. working from the inside, starting from 7. you would need to leave the list (7 is in a list by itself), then move up the list to six with cdr, together they make one cadr. then you are at the same position again with (6 7) instead of the 7. so you would need to leave the (6 7) list then move up a position to 5.

so it is just a repetition of `cadr` for each 'wrap'.

```
(define third '(1 (2 (3 (4 (5 (6 7)))))))  
(display (cadr (cadr (cadr (cadr (cadr (cadr (cadr third))))))))
```

Footnote 9. states:

Since nested applications of car and cdr are cumbersome to write, Lisp dialects provide abbreviations for them -- for instance,

(cadr <arg>) = (car (cdr <arg>))

The names of all such procedures start with `c` and end with `r`. Each `a` between them stands for a car operation and each `d` for a cdr operation, to be applied in the same order in which they appear in the name. The names `car` and `cdr` persist because simple combinations like `cadr` are pronounceable.

But, when I try `(cadaddr (list 1 3 (list 5 7) 9))` It returns an error: ;*Unbound variable: cadaddr*

There is a limit of operations one can concatenate in cxxr format (only four). **Common Lisp HyperSpec: CAR, CDR & etc.**)

Since `(cdaddr (list 1 3 (list 5 7) 9))` returns `(7)`, we can use `(car (cdaddr (list 1 3 (list 5 7) 9)))` to solve the first item.

I wrote a procedure to mimic the cxxr function. I don't like this implementation, but that was the way I found to

make the function work (I had some problems trying to write a procedure to compare char values, or even evaluating a whole string of a&d's. Using a raw sequence of 0&1's was too cryptic).

```
(define a 0)
(define d 1)
(define (cr items . ad-seq)
  (define ops (reverse ad-seq))
  (define (iter result oper)
    (if (null? oper)
        result
        (cond ((= (car oper) a) (iter (car result) (cdr oper)))
              ((= (car oper) d) (iter (cdr result) (cdr oper)))
              (else (error "Just cAr or cDr -- Char Not recognized" (car oper))))))
  (iter items ops))

;(1 3 (5 7) 9)

(cr (list 1 3 (list 5 7) 9) a d a d d)
;Value: 7
```

## Cool Stuff!

```
ctz

; create a procedure from a string (using the same format as "cadr", "caaddr", etc)
(define (cxxr str)
  (define (recur str lst)
    (let ((first (string-ref str 0))
          (rest (substring str 1)))
      (cond ((eq? first #\a)
             (car (recur rest lst)))
            ((eq? first #\d)
             (cdr (recur rest lst)))
            ((eq? first #\r)
             lst)
            (else (error "Unrecognizable symbol:" str)))))

  (lambda (lst)
    (if (eq? (string-ref str 0) #\c)
        (recur (substring str 1) lst)
        (error "Unrecognizable symbol:" str)))))

; find an atom a in the list l and give the string representing the procedure to pick a
(define (find a l)
  (define (searcher l)
    (cond ((null? l) #f)
          ((not (pair? l))
           (if (eq? a l)
               ""
               #f))
          (else (let ((a (searcher (car l)))
                     (d (searcher (cdr l))))
                 (cond (a (string-append a "a"))
                       (d (string-append d "d"))
                       (else #f))))))
    (string-append "c" (searcher l) "x")))

#| tests:
> (find 7 '(1 3 (5 7) 9))
"cadaddr"
> ((cxxr "cadaddr") '(1 3 (5 7) 9))
7
> (find 7 '(1 (2 (3 (4 (5 (6 7)))))))
"cadadadadadadr"
> ((cxxr "cadadadadadadr") '(1 (2 (3 (4 (5 (6 7)))))))
7
|#
```

# sicp-ex-2.26

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.25) | Index | Next exercise (2.27) >>

jz

```
(define x (list 1 2 3))
(define y (list 4 5 6))
```

(append x y) prints

```
(1 2 3 4 5 6)
```

because we're just building the list.

(cons x y) prints

```
((1 2 3) 4 5 6)
```

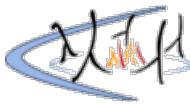
because it's a list of 4 elements, where the first element is also a list. Note that (cons a (list b c d)) is the same as (list a b c d).

(list x y) prints

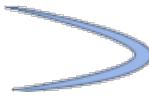
```
((1 2 3) (4 5 6))
```

because this is a list of two separate elements.

Last modified : 2010-02-09 08:44:04  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-2.27



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.26) | Index | Next exercise (2.28) >>

```
jz
;; A value for testing.
(define x (list (list 1 2) (list 3 (list 4 5)))))

;; My environment doesn't have nil.
(define nil '())

;; Here's reverse for reference:
(define (reverse items)
  (define (rev-imp items result)
    (if (null? items)
        result
        (rev-imp (cdr items) (cons (car items) result))))
  (rev-imp items nil))

;; Usage:
(reverse x)

;; Deep reverse. Same as reverse, but when adding the car to the
;; result, need to check if the car is a list. If so, deep reverse
;; it.

;; First try:
(define (deep-reverse items)
  (define (deep-rev-imp items result)
    (if (null? items)
        result
        (let ((first (car items)))
          (deep-rev-imp (cdr items)
                        (cons (if (not (pair? first))
                                  first
                                  (deep-reverse first))
                              result)))))

  (deep-rev-imp items nil))

;; Usage:
(deep-reverse x)

;; Works, but it's a bit hard to read? Refactoring:

(define (deep-reverse-2 items)
  (define (deep-rev-if-required item)
    (if (not (pair? item))
        item
        (deep-reverse-2 item)))
  (define (deep-rev-imp items result)
    (if (null? items)
        result
        (deep-rev-imp (cdr items)
                      (cons (deep-rev-if-required (car items))
                            result)))))

  (deep-rev-imp items nil))

;; Usage:
(deep-reverse-2 x)
```

Here's Eli Bendersky's code, translated into Scheme. It's pretty sharp, and better than my own since it's more concise:

```
(define (eli-deep-reverse lst)
  (cond ((null? lst) nil)
        ((pair? (car lst))
         (append
          (eli-deep-reverse (cdr lst))
          (list (eli-deep-reverse (car lst)))))))
```

```

        (else
          (append
            (eli-deep-reverse (cdr lst))
            (list (car lst)))))

(eli-deep-reverse x)

```

This works for me:

```

(define (deep-reverse x)
  (if (pair? x)
      (append (deep-reverse (cdr x))
              (list (deep-reverse (car x)))))
      x))

```

A solution that uses `reverse` to do the work:

```

(define (deep-reverse t)
  (if (pair? t)
      (reverse (map deep-reverse t))
      t))

```

 Another solution without append

```

(define (deep-reverse items)
  (define (iter items result)
    (if (null? items)
        result
        (if (pair? (car items))
            (let ((x (iter (car items) ())))
              (iter (cdr items) (cons x result)))
            (iter (cdr items) (cons (car items) result))))
        (iter items ()))))

```

 Solution that is a simple modification of reverse

```

(define (deep-reverse tree)
  (define (iter t result)
    (cond ((null? t) result)
          ((not (pair? (car t)))
           (iter (cdr t) (cons (car t) result)))
          (else
            (iter (cdr t) (cons (deep-reverse (car t)) result))))))
  (iter tree '()))

#|
> (deep-reverse '((1 2) (3 4)))
'((4 3) (2 1))
> (deep-reverse '(1 2 (3 4) 5 (6 (7 8) 9) 10))
'(10 (9 (8 7) 6) 5 (4 3) 2 1)
>
|#
>>>

```



there is another solution. it may be simpler.

```

(define (deep-reverse li)
  (cond ((null? li) '())
        ((not (pair? li)) li)
        (else (append (deep-reverse (cdr li))
                      (list (deep-reverse (car li)))))))

```

Daniel-Amariei

Took me a while to implement it without append and reverse.

```

(define (deep-reverse L)
  (define (rev L R)
    (cond ((null? L) R)
          ((not (pair? (car L))) (rev (cdr L)
                                         (cons (car L) R)))
          (else (rev (cdr L)
                      (cons (rev (car L) '())
                            R)))))

  (rev L '()))

(define x '((1 2) (3 4)))
(deep-reverse x) ;; ((4 3) (2 1))

```

atrika Solution with no use of `append`. Would be nice to have a full iterative process, but this problem is naturally recursive.

```

(define (deep-reverse l)
  (define (update-result result picked)
    (cons (if (pair? picked) (deep-reverse picked) picked) ;; recursive process
          result))

  (define (iter source result)
    (if (null? source)
        result
        (iter (cdr source) (update-result result (car source)))))

  (iter l '()))

;; testing
(deep-reverse '(1 2 (a b c (d1 d2 d3)) 4)) ;; returns '(4 ((d3 d2 d1) c b a) 2 1)

```

adam My solution which is very similar to the original.

```

(define (deep-reverse items)
  (define (try-deep item)
    (if (not (list? item))
        item
        (iter item '())))

  (define (iter old new)
    (if (null? old)
        new
        (iter (cdr old)
              (cons (try-deep (car old)) new)))))

  (iter items '()))

;; Testing
(define x (list (list 1 2) (list 3 (list 4 5)) (list (list 2 3) 3)))
;; ((1 2) (3 (4 5)) ((2 3) 3))
(deep-reverse x)
;; ((3 (3 2)) ((5 4) 3) (2 1))

```

yves I though I'll found my version. Very close to some version here tough, but using map for clarity. It looks like it is working. May I be wrong somewhere?

```

(define (deep-reverse tree)
  (cond ((null? tree) nil)
        ((not (pair? tree)) tree)
        (else (map deep-reverse (reverse tree)))))

(display (deep-reverse (list (list 1 2) (list 3 4))))
(newline)
(display (deep-reverse (list (list 1 (list 5 6)) (list 3 4)))))
(newline)

```

LambdaDef

The most briefly solution

This solution only works with list of lists

```
(define (deep-reverse l)
  (reverse (map reverse l)))
```

DeepDolphin

```
(define (deep-reverse lst)
  (if (not (pair? lst))
      lst
      (append (deep-reverse (cdr lst))
              (list (deep-reverse (car lst))))))

;;Testing
(define x (list (list 1 2) (list 3 (list 4 (list 5 6 7) (list 8 9 10) 11))))
(deep-reverse (deep-reverse x))
;; ((1 2) (3 (4 (5 6 7) (8 9 10) 11)))
```

joshwarrior

```
(define (reverse item)
  (define (reverse-iter item result)
    (if (null? item)
        result
        (reverse-iter (cdr item) (cons (car item) result))))
  (reverse-iter item nil))

(define (deep-reverse item)
  (define (deep-reverse-iter item result)
    (cond ((null? item) result)
          ((pair? (car item)) (deep-reverse-iter (cdr item) (cons (deep-reverse (car item)) result)))
          (else (deep-reverse-iter (cdr item) (cons (car item) result)))))
  (deep-reverse-iter item nil))

;;Testing
> (define x (list (list 1 2) (list 3 4)))
> x
(mcons (mcons 1 (mcons 2 '())))
  (mcons (mcons 3 (mcons 4 '())))
  '())
> (reverse x)
(mcons (mcons 3 (mcons 4 '())))
  (mcons (mcons 1 (mcons 2 '())))
  '())
> (deep-reverse x)
(mcons (mcons 4 (mcons 3 '())))
  (mcons (mcons 2 (mcons 1 '())))
  '())
> (deep-reverse (list 1 2 3))
(mcons 3 (mcons 2 (mcons 1 '())))
> (deep-reverse (list (list 1 2 3)))
(mcons 3 (mcons 2 (mcons 1 '())))
'()
```

ybsh

My first try: this is the same solution as joshwarrior's, except that the last two of cond's operands are reversed.

```
(define (deep-reverse l)
  (define (rev l res)
    (cond ((null? l) res)
          ((not (pair? (car l))) (rev (cdr l)
                                         (cons (car l)
                                               res)))
          (else (rev (cdr l)
                     (cons (rev (car l)
                               '())
                               res)))))

  (rev l '()))
```

Then I realized that car'ing before pair? was unnecessary, and here's my final answer.

```
(define (deep-reverse l)
  (define (rev l res)
    (cond ((null? l) res)
          ((not (pair? l)) l)
          (else (rev (cdr l)
                     (cons (rev (car l)
                               '())
                               res)))))
```

```
             res)))))  
(rev l '()))
```

I think this is better in terms of simplicity. This solution is also efficient because it does not use append or any operation that travels down the intermediate list from its head every time the function is called.

zerol

I think the solution can be much shorter by using map and list?.

```
(define (deep-reverse x)  
  (if (list? x)  
      (reverse (map deep-reverse x))  
      x))
```

tf3

This method is a very slight and (in my opinion) intuitive alteration of the reverse procedure. Instead of cons'ing the car of z to result (as in the original reverse procedure), the reverse of car of z is cons'd to the result.

```
(define (deep-reverse items)  
  (define (iter-reverse z result)  
    (cond ((null? z) result)  
          ((not (pair? z)) z)  
          (else (iter-reverse (cdr z) (cons (iter-reverse (car z) '()) result))))  
    ))  
  (iter-reverse items '())  
)
```

I felt that this solution was very 'conforming' to the spirit of doing things the Scheme way, and added minimum complexity over the vanilla version of reverse, and therefore might be a useful addition here

chessweb

My first contribution to this wiki

```
(define (deep-reverse l)  
  (define atom?  
    (lambda (x)  
      (and (not (pair? x)) (not (null? x)))))  
  (define (reverse l)  
    (if (null? (cdr l))  
        l  
        (append (reverse (cdr l)) (list (car l)))))  
  (cond  
    ((null? l) nil)  
    ((pair? (car l)) (append (deep-reverse (cdr l))  
                             (list (deep-reverse (car l)))))  
    ((atom? (car l)) (append (deep-reverse (cdr l))  
                             (list (car l))))  
    (else (reverse l))))  
  
(deep-reverse '(13 14 (1 2 3) ((3 4) (1 2)) (6 (7 8 9) (7))))
```

results in

```
((((7) (9 8 7) 6) ((2 1) (4 3)) (3 2 1) 14 13)
```

chessweb

which can be simplified

```
(define (deep-reverse l)  
  (cond  
    ((null? l) nil)  
    ((pair? (car l)) (append (deep-reverse (cdr l))  
                             (list (deep-reverse (car l)))))  
    (else (append (deep-reverse (cdr l))  
                  (list (car l))))))
```

rwitak

(first-timer here)

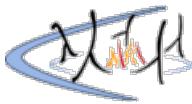
I think zerol's solution wins the cake for simplicity, readability, clarity AND application of newly learned material.

But here is mine, as it is somewhat different from the others:

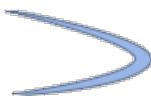
```
(define (deep-reverse items)
  (if (pair? items)
      (reverse (cons (deep-reverse (car items))
                     (deep-reverse (cdr items))))))
  items))
```

tch

I think using list? as condition is better than pair? since list? is strict than pair? and the element of list could be just pair but not list. Consider a case like (list (cons 1 2) (cons 3 4)). Of course, pair? is enough for this excercise.



# sicp-ex-2.28



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.27) | Index | Next exercise (2.29) >>

```
;; Fringe.

(define my-tree (list 1 (list 2 (list 3 4) (list 5 6)) (list 7 (list 8)))))

;; First try. If the current element is a leaf, cons it to the fringe
;; of the rest. If the current element is a tree, cons the fringe if
;; it to the fringe of the rest. .... Won't work because the bad line
;; indicated won't build a flat list, it will always end in nested
;; cons.
(define (fringe tree)
  (define nil '())
  (if (null? tree)
      nil
      (let ((first (car tree)))
        (if (not (pair? first))
            (cons first (fringe (cdr tree)))
            ;; bad line follows:
            (cons (fringe first) (fringe (cdr tree)))))))

(fringe my-tree)

;; Second try.
;; Need to store fringe of the cdr, then add the fringe of the car to
;; that.
(define (fringe tree)
  (define nil '())

  (define (build-fringe x result)
    (cond ((null? x) result)
          ((not (pair? x)) (cons x result))
          (else (build-fringe (car x)
                               (build-fringe (cdr x) result)))))

  (build-fringe tree nil))

;; Usage:
(fringe my-tree)
```

Is there a better way to do this? Specifically, is there a way to do this only in terms of fringe, without creating another function?

Yes, there is a better way ... just use append instead of cons in version 1.

```
(define (fringe tree)
  (define nil '())
  (if (null? tree)
      nil
      (let ((first (car tree)))
        (if (not (pair? first))
            (cons first (fringe (cdr tree)))
            (append (fringe first) (fringe (cdr tree)))))))

;; Usage
(fringe my-tree)
(fringe (list 3))
(fringe 3) ;; fails, as it should.
```

This takes an alternate form later in the book (enumerate-tree in a future chapter), and Eli Bendersky's answer was also slightly different than mine. They didn't bother with doing all the (let ((first (car tree)...)) nonsense:

```
(define (fringe tree)
  (define nil '())
  (cond ((null? tree) nil)
        ((not (pair? tree)) (list tree)))
```

```

    (else (append (fringe (car tree)) (fringe (cdr tree))))))
(fringe my-tree)

```

A much cleaner solution, with the caveat (if it can be called such) that this function works even if passed a non-tree (eg, (fringe 3) => 3), while the previous fails as expected. Again, not really a valid caveat.

A solution which I found that happens to be similar to the one from enumerate-tree, but which fails if passed a non-list.

```

(define (fringe x)
  (define (fringe-recur x result)
    (cond ((null? x)
           result)
          ((not (pair? (car x)))
           x)
          (else
            (fringe-recur (cdr x) (append result (fringe-recur (car x) (list)))))))
  (fringe-recur x (list)))

```

rnsmmit

### iterative solution

```

;;take all leafs as a list
(define (fringe items)
  (define (fringe-iter items result)
    (cond ((null? items)
           result)
          ((pair? items)
           (fringe-iter (car items)
                       (fringe-iter (cdr items) result)))
          (else (cons items result))))
  (fringe-iter items nil))

(fringe (list (list 1 2) (list 3 4 (list 5 (list 6)))))

```

VladimirF

The versions with append fail for lists with n~1000 for me. The version with just cons works nicely up to n~100000 with guile.

holub

### my way:

```

(define (fringe items)
  (define (fringe-iter source dist)
    (if (null? source)
        dist
        (fringe-iter (cdr source)
                    (append dist
                            (let ((scar (car source)))
                              (if (pair? scar)
                                  (fringe scar)
                                  (list scar)))))))
  (fringe-iter items (list)))

```

Daniel-Amariei

Puts ones mind to work.

```

;; construct the list from the right of the tree, to the left
(define (fringe T)
  (define (iter T R)
    (cond ((null? T) R)
          ((not (pair? T)) (cons T R))
          (else (iter (car T)
                      (iter (cdr T) R)))))

  (iter T '()))

(define x '(1 (2 (3 4)))

(fringe x) ; (1 2 3 4)
(fringe (list x x)) ; (1 2 3 4 1 2 3 4)
(fringe '(2)) ; 2

```

Alex Gunnarson

I did this late at night kind of accidentally. I tried to put the least amount of thought into it possible and I messed around with the code without really thinking, shoved it in a REPL, and voila! the problem solved itself. For comparison's sake, I spent a grand total of 5 minutes on this but a few hours on Problem 2.27. Wish I could have solved 2.27 as easy.

```
(define (fringe L)
  (define (fringe-help L result)
    (if (null? L) ; if at end of the branch
        result
        (if (list? L) ; if the element is a list, not a number
            (fringe-help (car L) ; left branch
                         (fringe-help (cdr L) result)) ; right branch
            (cons L result)))) ; otherwise gather numbers into a list
  (fringe-help L '()))

(fringe '((1 2) (3 4) (5 6 7 (8 9))))
;; (1 2 3 4 5 6 7 8 9)
```

Adam Stanton

Here is my solution. I'm eager to find a better solution that performs better!

```
(define (fringe tree)
  (define (iter x new)
    (if (null? x)
        new
        (let ((first (car x)))
          (if (list? first)
              (append (iter first new) (iter (cdr x) new))
              (cons first (iter (cdr x) new))))))

  (iter tree '()))

;; Testing
(fringe (list (list (list 1 2 3 19 283 38) 2 3 2) (list 2 3 (list 217 382 1827) 2 187
(list 2838)) 2 1 2 (list 2 (list 3 (list 3)) 23 2 1 238)))

(1 2 3 19 283 38 2 3 2 2 3 217 382 1827 2 187 2838 2 1 2 2 3 3 23 2 1 238)
```

Mathieu Bordere

And another one ... (thanks for the testcase Adam!)

```
(define (fringe tree)
  (cond ((null? tree) '())
        ((list? (car tree)) (append (fringe (car tree))
                                      (fringe (cdr tree))))
        (else (cons (car tree) (fringe (cdr tree))))))

;; Testing
(fringe (list (list (list 1 2 3 19 283 38) 2 3 2) (list 2 3 (list 217 382 1827) 2 187
(list 2838)) 2 1 2 (list 2 (list 3 (list 3)) 23 2 1 238)))

(1 2 3 19 283 38 2 3 2 2 3 217 382 1827 2 187 2838 2 1 2 2 3 3 23 2 1 238)
```

zhenhuaa

A simple version use number?

```
(define (fringe x)
  (cond ((null? x) x)
        ((number? x) (list x))
        (else (append (fringe (car x))
                      (fringe (cdr x))))))
```

vpraid

This is my iterative solution. It doesn't use append and it doesn't rely on call stack. Instead, it uses its own stack. This should work even for extremely large trees.

```
(define (reverse x)
  (define (iter x acc)
    (if (null? x)
        acc
```

```

        (iter (cdr x)
              (cons (car x) acc)))
    (iter x nil))

(define (fringe x)
  (define (collect stack acc)
    (if (null? stack)
        acc
        (let ((top (car stack)))
          (cond ((null? top) (collect (cdr stack) acc))
                ((not (pair? top)) (collect (cdr stack) (cons top acc)))
                (else (collect (cons (car top)
                                      (cons (cdr top) (cdr stack)))
                               acc))))))
  (reverse (collect (list x) nil)))

```

ly

short code

```

(define (fringe x)
  (cond ((null? x) #nil)
        ((pair? x)
         (append (fringe (car x))
                 (fringe (cdr x))))
        (else (list x))))

```

musiXelect

My solution. It doesn't do anything special but why wouldn't I share it?

```

(define nil '())
(define (fringe ls)
  (define (helper ls result)
    (cond ((null? ls) result)
          ((not (pair? ls)) (cons ls result))
          (else (append (helper (car ls) result) (helper (cdr ls) result))))))
  (helper ls nil))

```

Nico de Vreeze

Not using append and/or helper proc:

```

(define (fringe items)
  (cond ((not (pair? items)) items)
        ((pair? (car items)) (fringe (cons (caar items) (cons (cdar items) (cdr
items))))))
        ((null? (car items)) (fringe (cdr items)))
        (else (cons (car items) (fringe (cdr items))))))

;; Also works with empty sub-lists:
(fringe '(1 2 3 (4 5) 6 () (((7 (8 9) ()) 10) 11) 12))
;;Value: (1 2 3 4 5 6 7 8 9 10 11 12)

```

rohitkg98

Completely iterative solution but it reverses the contents The idea is to maintain three state variables *first*, *rest*, *res* Trace of ((1 2) (3 4)):

first	rest	res
nil	((1 2) (3 4))	nil
(1 2)	((3 4))	nil
1	(2 (3 4))	nil
nil	(2 (3 4))	(1)
2	((3 4))	(1)
nil	((3 4))	(2 1)
(3 4)	nil	(2 1)
3	(4)	(2 1)
nil	(4)	(3 2 1)
4	nil	(3 2 1)
nil	nil	(4 3 2 1)

So instead of call stack we maintain info in rest.

```
(define (fringe-iter-reverse items)
  (define (iter first rest res)
    (cond ((and (null? first) (null? rest)) res)
          ((null? first) (iter (car rest) (cdr rest) res))
          ((pair? first) (iter (car first) (cons (cdr first) rest) res))
          (else (iter (car rest) (cdr rest) (cons first res))))))
  (iter '() items '()))
```

madhur95

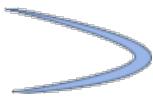
Inspired by somarl's solution in 2.22 of using a higher order function to store the leaves to avoid reversing the final answer. Maybe this approach can be combined with rohitkg98's approach to get complete iterative solution.

```
(define (fringe lst)
  ; iter succesively reduces the inputted list by cdr ing it
  (define (iter lst f is-outer-list?)
    ; f keeps record of the leaves using cons and successive application of itself
    ; list of leaves is formed by applying f to nil when finally original list has
    ; been cdr ed into empty.
    ; is-outer-list? ensures nil is applied only once to make final list, is #t for
    ; outermost list and #f for any sub-lists found within the original list
    (cond ((null? lst) (if is-outer-list?
                           (f lst)
                           f))
          ; edge case
          ((not (pair? lst)) lst)
          ; if first element is a leaf, add it to the function
          ((not (pair? (car lst)))
           (iter (cdr lst)
                 (lambda (x) (f (cons (car lst) x)))
                 is-outer-list?)))
          ; otherwise add all the leaves of first element to the function
          (else (iter (cdr lst)
                      (iter (car lst) f #f)
                      is-outer-list?)))
          ))
    (iter lst (lambda (x) x) #t))

(fringe (list 1 (list 1 2) 3 (list 8 9 (list 10 11))))
; outputs -> '(1 1 2 3 8 9 10 11)
```



# sicp-ex-2.29



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.28) | Index | Next exercise (2.30) >>

2DSharp

Making the constructors and selectors first for building the bigger solution on.

Version one using lists

```
(define (make-mobile left right)
  (list left right))
(define (left-branch mobile)
  (car mobile))
(define (right-branch mobile)
  (car (cdr mobile)))

(define (make-branch length structure)
  (list length structure))

(define (branch-length branch)
  (car branch))
(define (branch-structure branch)
  (car (cdr branch)))
```

Version two using cons instead of lists

```
(define (make-mobile left right)
  (cons left right))

(define (left-branch mobile)
  (car mobile))
(define (right-branch mobile)
  ; Have to use cdr at this point
  (cdr mobile))

(define (make-branch length structure)
  (cons length structure))

(define (branch-length branch)
  (car branch))
(define (branch-structure branch)
  (cdr branch))
```

Rest of the problem

```
; Finding the total weight of a binary mobile
; Using wishful thinking to recurse the addition of left-branch and right-branch mobiles

(define (total-weight mobile)
  (cond ((null? mobile) 0)
        ((not (pair? mobile)) mobile)
        (else (+ (total-weight (branch-structure (left-branch mobile)))
                  (total-weight (branch-structure (right-branch mobile)))))))

; Test
(define a (make-mobile (make-branch 2 3) (make-branch 2 3)))
(total-weight a) ; 6

(define (torque branch)
  (* (branch-length branch) (total-weight (branch-structure branch)))))

; Finally to check if the torques of both sides are equal
; And if the sub-mobiles are balanced using recursion

(define (balanced? mobile)
  (if (not (pair? mobile))
      true
      (and (= (torque (left-branch mobile)) (torque (right-branch mobile)))
            (balanced? (branch-structure (left-branch mobile)))
            (balanced? (branch-structure (right-branch mobile))))))

; Test
```

```

(define d (make-mobile (make-branch 10 a) (make-branch 12 5)))
;; Looks like: ((10 ((2 3) (2 3))) (12 5))

(balanced? d) ;; #t

```

The test cases were used from Bill the Lizard's blog on SICP

Rather Iffy

Use the same recursion over the construction of the mobile for defining the function 'total-weight' as; well as the predicate 'balanced?'. Take advantage of the fact that a variable has no type. The variable 'm' can in different calls take as a value a mobile, a branch or a number or empty list.

```

; Definition of constructors and selectors

(define (make-mobile left right)
  (list left right))
(define (left-branch mobile)
  (car mobile))
(define (right-branch mobile)
  (car (cdr mobile)))

(define (make-branch length structure)
  (list length structure))
(define (branch-length branch)
  (car branch))
(define (branch-structure branch)
  (car (cdr branch)))

;; Redefinition of constructors and selectors

(define (make-mobile left right)
  (cons left right))
(define (left-branch mobile)
  (car mobile))
(define (right-branch mobile)
  (cdr mobile))

(define (make-branch length structure)
  (cons length structure))
(define (branch-length branch)
  (car branch))
(define (branch-structure branch)
  (cdr branch))

(define (total-weight m)
  (cond ((null? m) 0)
        ((not (pair? m)) m)
        (else (+ (total-weight (branch-structure (left-branch m)))
                  (total-weight (branch-structure (right-branch m)))))))

(define m1 (make-mobile
            (make-branch 4 6)
            (make-branch 5
                        (make-mobile
                          (make-branch 3 7)
                          (make-branch 9 8)))))

;;      4   |   5
;;      +---+---+
;;      6       3   |   9
;;                  +---+---+
;;                  7       8

; (total-weight m1)
; Value: 21

(define (balanced? m)
  (cond ((null? m) #t)
        ((not (pair? m)) #t)
        (else
          (and (= (* (branch-length (left-branch m))
                    (total-weight (branch-structure (left-branch m))))
                  (* (branch-length (right-branch m))
                    (total-weight (branch-structure (right-branch m)))))

                (balanced? (branch-structure (left-branch m)))
                (balanced? (branch-structure (right-branch m)))))))

(define m2 (make-mobile
            (make-branch 4 6)
            (make-branch 2
                        (make-mobile
                          (make-branch 5 8)
                          (make-branch 3 7)))))

; (balanced? m2)
; Value: #f

```

```

        (make-branch 10 4)))))

;;
  4 | 2
+---+---+
6   5 |    10
+---+-----+
8           4

(balanced? m2)
;Value: #t
(balanced? m1)
;Value: #f

```

seek

My solution for c. consists of only one tree traversal instead of recursive tree traversal due to 'total-weight' procedure in every node.

```

(define (make-mobile l r) (list l r))
(define (make-branch len struct) (list len struct))

;; a.
(define (left-branch m) (car m))
(define (right-branch m) (cadr m))
(define (branch-length b) (car b))
(define (branch-structure b) (cadr b))

;; b.
(define (total-weight m)      ; m should be mobile
  (if (not (pair? m))
      m
      (+ (branch-structure (left-branch m))
          (branch-structure (right-branch m)))))

;; c.
;; Defined [ sum-balanced?-pair :: mobile |-> (sum, balanced?) ] instead of using total-
;; weight to reduce recursions.
(define (balanced? m)      ; m should be mobile
  (define (sum-balanced?-pair m)
    (if (not (pair? m))
        (cons m #t)
        (let ((left (sum-balanced?-pair (branch-structure (left-branch m))))
              (right (sum-balanced?-pair (branch-structure (right-branch m)))))
          (cons (+ (car left)
                    (car right))
                (and (cdr left)
                     (cdr right)
                     (= (* (branch-length (left-branch m)) (car left))
                        (* (branch-length (right-branch m)) (car right)))))))
        (cdr (sum-balanced?-pair m)))))

;; d.
;; Modifying selectors is sufficient.

;; test cases from http://community.schemewiki.org/?sicp-ex-2.29
(define a (make-mobile (make-branch 2 3) (make-branch 2 3)))
(total-weight a)

(define d (make-mobile (make-branch 10 a) (make-branch 12 5)))
(balanced? d)

```

jay

My solution for 'balanced?' function. Aoivd using total-weight to reduce recursive tree, also only use one return value.

```

(define (balanced? mb)
  (define (balanced-rec? mb)
    ;; balance? returns #f if mb is not balanced
    ;;           returns its weight if mb is balanced
    (cond ((null? mb) 0)
          ((number? mb) mb)
          ((pair? mb)
            (let ((left (balanced-rec? (branch-structure (left-branch mb)))))
              (right (balanced-rec? (branch-structure (right-branch mb))))))
            ;; because (and number #t) = #t
            ;;           (and #f #t) = #f
            (if (or (not (and left #t)) (not (and right #t)))
```

```

;; if left or right is not balanced, return #f
#f
;; else calculate torque
(if (= (* left (branch-length (left-branch mb)))
      (* right (branch-length (right-branch mb))))
    ;; if balanced return its weight
    (+ left right)
    ;; else return #f
    #f))))))
(and (balanced-rec? mb) #t))

```

tf3

This solution touches each branch once and goes the full depth until it hits a node which has only weights on its branches. It returns a list of two values, the first one tells if the node is balanced and the second one captures the weight on the node. So every time the function returns I am able to retrieve the weights of the children (i.e. branches) and calculating torque is only a matter of multiplication. No separate procedure calls for torque.

```

(define (make-mobile left right)
  (list left right)
)
(define (make-branch length structure)
  (list length structure)
)

(define (left-branch mobile) (car mobile))
(define (right-branch mobile) (cadr mobile))
(define (branch-length branch) (car branch))
(define (branch-structure branch) (cadr branch))

(define (total-weight mobile)
  (cond ((and (not (pair? (branch-structure (left-branch mobile))))
              (not (pair? (branch-structure (right-branch mobile))))))
         (+ (branch-structure (left-branch mobile))
            (branch-structure (right-branch mobile))))
        ((not (pair? (branch-structure (left-branch mobile))))))
         (+ (branch-structure (left-branch mobile))
            (total-weight (branch-structure (right-branch mobile)))))
        ((not (pair? (branch-structure (right-branch mobile))))))
         (+ (total-weight (branch-structure (left-branch mobile)))
            (branch-structure (right-branch mobile))))
        (else
          (+ (total-weight (branch-structure (left-branch mobile)))
             (total-weight (branch-structure (right-branch mobile))))))
      )
    )

(define (balanced? mbl)
  (define (balanced-aux? mobile)
    (cond
      ((and (not (pair? (branch-structure (left-branch mobile))))))
       (not (pair? (branch-structure (right-branch mobile)))))

      (list (= (* (branch-length (left-branch mobile)) (branch-structure (left-branch mobile)))
                (* (branch-length (right-branch mobile)) (branch-structure (right-branch mobile))))
             (+ (branch-structure (left-branch mobile)) (branch-structure (right-branch mobile)))))

      ((not (pair? (branch-structure (left-branch mobile))))))
       (let ((balR (balanced-aux? (branch-structure (right-branch mobile)))))

         (list (and (car balR)
                    (= (* (branch-length (left-branch mobile)) (branch-structure (left-branch mobile)))
                       (* (branch-length (right-branch mobile)) (cadr balR))))
                  (+ (branch-structure (left-branch mobile)) (cadr balR)))))

      ((not (pair? (branch-structure (right-branch mobile))))))
       (let ((balL (balanced-aux? (branch-structure (left-branch mobile)))))

         (list (and (car balL)
                    (= (* (branch-length (right-branch mobile)) (branch-structure (right-branch mobile)))
                       (* (branch-length (left-branch mobile)) (cadr balL))))
                  (+ (cadr balL) (branch-structure (right-branch mobile)))))))

      (else
        (let ((balL (balanced-aux? (branch-structure (left-branch mobile)))))

          (balR (balanced-aux? (branch-structure (right-branch mobile)))))

          (list (and (car balL) (car balR)
                     (= (* (branch-length (left-branch mobile)) (cadr balL)))
                        (+ (cadr balL) (branch-structure (right-branch mobile)))))))


```

```

        (= (* (branch-length (left-branch mobile)) (cadr ballL))
              (* (branch-length (right-branch mobile)) (cadr balR))))
        (+ (cadr ballL) (cadr balR))))
    )
  (car (balanced-aux? mbl))
)

```

master

Here's my solution. I prefer to factor out the code that calculates the branch weight from `total-weight`, which makes both `weight` and `total-weight` much cleaner and easier to understand. One could argue that the name `total-weight` then isn't appropriate anymore and really we have two functions, which calculate either the branch weight or mobile weight. They are mutually recursive but that's because the data structure is mutually recursive in a sense, mobiles contain branches which can contain other mobiles, etc.

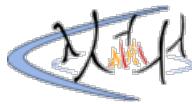
```

(define (weight branch)
  (let ((submobile (branch-structure branch)))
    (if (not (mobile? submobile))
        submobile
        (total-weight submobile)))

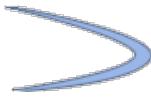
(define (total-weight mobile)
  (let ((left (left-branch mobile))
        (right (right-branch mobile)))
    (+ (weight left) (weight right)))

(define (balanced? mobile)
  (define (torque branch)
    (* (branch-length branch) (weight branch)))
  (if (not (mobile? mobile))
      #t
      (let ((left (left-branch mobile))
            (right (right-branch mobile)))
        (and (= (torque left) (torque right))
             (balanced? (branch-structure left))
             (balanced? (branch-structure right)))))))

```



# sicp-ex-2.30



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.29) | Index | Next exercise (2.31) >>

jz

```
(define my-tree (list 1 (list 2 (list 3 4) 5) (list 6 7)))  
  
;; Defining directly:  
(define (square-tree tree)  
  (define nil '()) ;; my env lacks nil  
  (cond ((null? tree) nil)  
        ((not (pair? (car tree)))  
         (cons (square (car tree)) (square-tree (cdr tree))))  
        (else (cons (square-tree (car tree))  
                     (square-tree (cdr tree))))))  
  
;; The above works, but there's no need to rip the tree apart in the (not  
;; ...) cond branch, since square-tree takes the tree apart for us in  
;; the else branch if we do one more recurse. The following is  
;; better:  
(define (sq-tree-2 tree)  
  (define nil '())  
  (cond ((null? tree) nil)  
        ((not (pair? tree)) (square tree))  
        (else (cons (sq-tree-2 (car tree))  
                     (sq-tree-2 (cdr tree))))))  
  
;; By using map:  
(define (sq-tree-with-map tree)  
  (define nil '())  
  (map (lambda (x)  
          (cond ((null? x) nil)  
                ((not (pair? x)) (square x))  
                (else (sq-tree-with-map x))))  
    tree))  
  
;; Usage  
(square-tree my-tree)  
(sq-tree-2 my-tree)  
(sq-tree-with-map my-tree)  
  
;; Originally, had the following for the else in sq-tree-with-map:  
;;  
;;  (else (cons (sq-tree-with-map (car x))  
;;               (sq-tree-with-map (cdr x)))))  
;;  
;; defining the else clause this way raised an error:  
;;  
;;  The object 3, passed as an argument to map, is not a list. etc.  
;;  
;; my-tree had a primitive, 3, as one branch of the tree, so calling  
;; sq-tree-with-map eventually resolved to (sq-tree-with-map 3), which  
;; chokes. Anyway, it was too much work, since all that was needed  
;; was a call to map *that* tree.
```

bishboria

an alternative version of map but doesn't use cond and doesn't need to check for nil.

```
(define (square-tree tree)  
  (map (lambda (sub-tree)  
          (if (pair? sub-tree)  
              (square-tree sub-tree)  
              (square sub-tree)))  
    tree))
```

meteorgan

This problem is similar to the example "scale-tree". so we can solve it as following:

```
(define (square-tree1 tree)
  (cond ((null? tree) '())
        ((not (pair? tree)) (* tree tree))
        (else (cons (square-tree1 (car tree))
                     (square-tree1 (cdr tree))))))

(define (square-tree2 tree)
  (map (lambda (x)
          (if (pair? x)
              (square-tree2 x)
              (* x x)))
       tree))
```

master

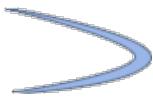
In my opinion, mapping a complex procedure over a list kind of undermines the conceptual simplicity usually gained from using `map`. I think it's much clearer what's going on in the following solution:

```
(define (square-tree tree)
  (if (not (pair? tree))
      (square tree)
      (map square-tree tree)))
```

It's pretty much the same idea but inside-out, instead of making a decision as to whether to recurse while mapping, we make a decision as to whether to map while recursing.



# sicp-ex-2.31



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.30) | Index | Next exercise (2.32) >>

jz Pretty nice.

```
(define (tree-map proc tree)
  (define nil '())
  (map (lambda (subtree)
    (cond ((null? subtree) nil)
          ((not (pair? subtree)) (proc subtree))
          (else (tree-map proc subtree))))
    tree))

;; Usage:
(define my-tree (list (list 1 2) 3 (list 4 5)))
(tree-map square my-tree)
(tree-map (lambda (x) (+ x 1)) my-tree)
```

jwc

my way:

```
(define (tree-map proc tree)
  (cond ((null? tree) nil)
        ((pair? tree)
         (cons
          (tree-map proc (car tree))
          (tree-map proc (cdr tree))))
        (else (proc tree))))
```

bishboria

my way. For me this is more succinct than using cond and there's no need to check for nil explicitly (or define it):

```
(define (tree-map proc tree)
  (map (lambda (sub-tree)
    (if (pair? sub-tree)
        (tree-map proc sub-tree)
        (proc sub-tree))))
    tree))
```

akaucher

own map implementation without cons

```
(define (tree-map proc tree)
  (if (null? tree)
      '()
      (if (pair? tree)
          (cons (tree-map proc (car tree))
                (tree-map proc (cdr tree)))
          (proc tree))))
```

pritesh

using map

```
(define (tree-map f tree)
  (map (lambda (x)
    (if (pair? x)
        (tree-map f x)
```

```
        (f x)
      )
    tree
  )
```

master

As I mentioned in the previous exercise, I think the idea of mapping over a tree is better expressed in the following way (although admittedly it's slightly more complex in the case where the procedure being mapped is not a unary procedure):

```
(define (tree-map f tree)
  (if (not (pair? tree))
    (f tree)
    (map (lambda (subtree) (tree-map f subtree)) tree)))
```

---

Last modified : 2021-12-19 01:25:39  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-2.32

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.31) | Index | Next exercise (2.33) >>

jz

This would be a tough problem in other languages, but it's really terse in Scheme. The hard part is the logic, but it's pretty clear when you get your head around it.

The set of all subsets of a given set is the union of:

- the set of all subsets excluding the first number.
- the set of all subsets excluding the first number, with the first number re-inserted into each subset.

Example 1: given the set (3), the first bullet gives the subset (), and the second bullet gives (), (3).

Example 2: given the set (2, 3), the first bullet gives the subsets () and (3). The second bullet gives (2), (2, 3), (), and (3). Note that the first two subsets are the same as the last two subsets with 2 unioned into each subset.

*Note this logic is similar to the coin-counting problem in section 1.2.2.*

```
(define nil '())

(define (subsets s)
  (if (null? s)
      (list nil)    ;; initially had nil, always got () back!
      (let ((rest (subsets (cdr s))))
        (append rest (map (lambda (x) (cons (car s) x)) rest)))))

(subsets (list 1 2 3))
(subsets (list 1 2 3 4 5))
```

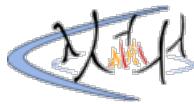
This problem becomes easy when you evolve the process manually:

```
(subsets '(1 2 3))
rest ← (subsets '(2 3))
rest ← (subsets '(3))
rest ← (subsets '())
'(())
  (append '() (map (...) '()))
  '()
  '(() (3))
  (append '() (3)) (map (...) '() (3)))
  '() (3) (2) (2 3))
  (append '() (3) (2) (2 3)) (map (...) '() (3) (2) (2 3)))
  '() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))
```

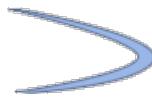
The problem now is to find the function  $(\lambda (x))$  to map, which has the characteristics:

'()	$\mapsto$	'((3))	given $s = '(3)$
'()	$\mapsto$	'((2) (2 3))	given $s = '(2 3)$
'()	$\mapsto$	'((1) (1 3) (1 2) (1 2 3))	given $s = '(1 2 3)$

Which is plainly the result of prepending the first item of S to each sublist X; that is, to cons the car of S onto each sublist. In Scheme parlance,  $(\lambda (x) (cons (car s) x))$ .



# sicp-ex-2.33



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.32) | Index | Next exercise (2.34) >>

jz

```
; ; Ex 2.33

;; Accumulates the result of the first and the already-accumulated
;; rest.
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))

(define nil '()) ; stupid environment ...

;; a. map

(define (my-map proc sequence)
  (accumulate (lambda (first already-accumulated)
               (cons (proc first) already-accumulated))
             nil
             sequence))

;; Test:

(my-map square (list))
(my-map square (list 1 2 3 4))

;; b. append

(define (my-append list1 list2)
  (accumulate cons
              list2
              list1))

;; Test:

	append (list 1 2 3) (list 4 5 6)) ; checking order.
(my-append (list 1 2 3) (list 4 5 6))

;; c. length

(define (my-length sequence)
  (accumulate (lambda (first already-acc)
               (+ 1 already-acc))
             0
             sequence))

;; Test:
(length (list 1 2 3 (list 4 5)))
(my-length (list 1 2 3 (list 4 5)))
```

atrika

```
(define (map p sequence)
  (accumulate (lambda (x y) (cons (p x) y)) '() sequence))

(define (append seq1 seq2)
  (accumulate cons seq2 seq1))

(define (length sequence)
  (accumulate (lambda (x y) (+ 1 y)) 0 sequence))
```

Nill

This map can also used in tree.

```
(define (map p sequence)
  (accumulate (lambda (x y)
    (cons (if (not (pair? x))
      (p x)
      (map p x))
        y))
    nil
    sequence))

;;Test:

(define a (list 1 (list 9 8) 3))
(map (lambda (x) (* x x)) a)
```

emj

IMO the trickiest thing about these accumulate problems is how accumulates works. To that end I like to remind myself by starting each problem with an expanded evaluation of accumulate, for a 2 item sequence:

```
;; accumulate for 2 item seq: (op (car seq) (op (car (cdr seq)) initial))

(define (append list1 list2)
  (accumulate cons list2 list1))

;; expands to:
;; (cons (car list1) (cons (car (cdr list1)) list2))
```

f1codz

It helped me to realize what accumulate translates to:

```
;; (accumulate op init (list 1 2 3))
;; (op 1 (op 2 (op 3 init)))
```

vs. what map translates to:

```
;; (map op (list 1 2 3))
;; (cons (op 1) (cons (op 2) (cons (op 3))))
```

and hence map in terms of accumulate would be:

```
(define (map op seq)
  (accumulate (lambda (next-ele mapped-seq)
    (cons (op next-ele) mapped-seq))
    '()
    seq))
```

# sicp-ex-2.34

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.33) | Index | Next exercise (2.35) >>

Taking the example of  $1 + 3x + 5x^3 + x^5$ , applying Horner's rule gives us  $((((1x + 0)x + 5)x + 0)x + 3)x + 1$ . Translated into prefix notation yields  $(+ (* (+ (* (+ (* (+ (* 1 x) 0) x) 5) x) 0) x) 3) x) 1$ , which reveals the recursive structure:

```
(+ (* (+ (* (+ (* (+ (* 1 x) 0) x) 5) x) 0) x) 3) x) 1)
(+ (* ..... x) 1)
(+ (* ..... x) 3)
etc.
```

Which indicates that the function to be used to accumulate will be  $(+ (* \text{higher-order terms} x) \text{coefficient})$ :

```
(define (horner-eval x coefficient-sequence)
  (accumulate (lambda (this-coeff higher-terms)
                (+ (* higher-terms x) this-coeff))
              0
              coefficient-sequence))

(horner-eval 2 (list 1 3 0 5 0 1))
⇒ 79
(horner-eval 2 (list 2 3 0 5 0 1)) ; Expect 1 greater than above.
⇒ 80
(horner-eval 2 (list 2 3 0 5 0 2)) ; Expect  $2^5 = 32$  greater than above.
⇒ 112
```

jz ; Accumulates the result of the first and the already-accumulated  
;; rest.  

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))

(define nil '()) ; stupid environment ...

(define (horner-eval x coefficient-sequence)
  (accumulate (lambda (this-coeff accum-sum)
                (+ this-coeff
                   (* x accum-sum)))
              0
              coefficient-sequence))

(horner-eval 2 (list 1 3 0 5 0 1))
```

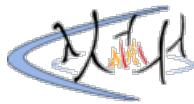
Rather Iffy ; I prefer recursion on the right because it makes it easier to grasp the  
recursion process.  

```
(define (horner-eval x coefficient-sequence)
  (accumulate (lambda (this-coeff higher-terms)
                (+ this-coeff (* higher-terms x)))
              0
              coefficient-sequence))
```

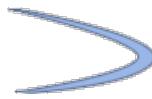
sateesh

this is a comment on solution posted by jz. @jz I think calling the variable as accum-sum is bit misleading, as the processing hasn't accumulated any sum yet .





# sicp-ex-2.35



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

[<< Previous exercise \(2.34\)](#) | [Index](#) | [Next exercise \(2.36\) >>](#)

Using recursion, it's possible to do it with map and without enumerate-tree the following way.

```
(define (count-leaves-recursive t)
  (accumulate + 0 (map (lambda (node)
    (if (pair? node)
        (count-leaves-recursive node)
        1))
  t)))
;; Usage
(count-leaves-recursive tree) ;; => 7
```

The previous solution counts the empty list as a leaf:

```
(count-leaves-recursive (quote (1 () () (( )) () ( () 2))))
;; Value: 8
```

This can easily be remedied by using more conditions:

```
(define (count-leaves-recursive t)
  (accumulate + 0
    (map
      (lambda (t)
        (cond ((null? t) 0)
              ((pair? t) (count-leaves-recursive t))
              (else 1)))
    t)))
;; which gives
(count-leaves-recursive '(1 2 () () (3 () ())))
;; Value: 3
```

amasad

For those who are confused about how the above is working consider the following example:

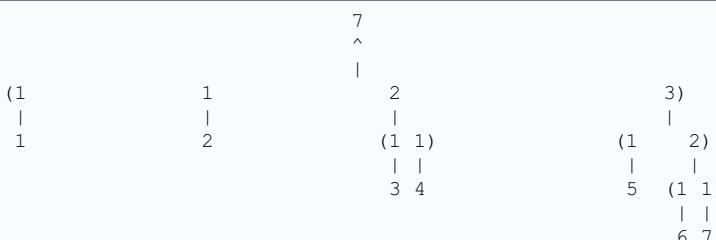
Having the following tree:

```
(define tree (list 1 2 (list 3 4) (list 5 (list 6 7))))
(count-leaves-recursive tree) ;; => 7
```

The map call in the function would flatten the tree into a list of the top level nodes with each one containing the count of its children nodes, if the node in it self is not a tree then 1 would be returned from the iterator passed to the map call.

Yielding (list 1 1 2 3) Which would be passed to the accumulator and a simple summation would be made.

The following illustrates the mapping on the tree, numbers would map to one and lists would map to its sum:



My solution is a "tomato-tomato" of above.

Identity-mapping and using recursion in proc.

```
(define (count-leaves t)
  (define proc (lambda (x y) (if (pair? x) (+ (count-leaves x) y)
                                    (+ 1 y))))
  (define init_val 0)
  (define seq (map (lambda (a) a) t))
  (accumulate proc init_val seq))
```

jz

I couldn't figure out a way of doing this with map without relying on the previously-defined enumerate-tree. enumerate-tree is used a lot in the chapter so it's probably what's intended.

```
; Accumulates the result of the first and the already-accumulated
; rest.
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))

(accumulate + 0 (list 1 2 3 4 5))
(accumulate cons nil (list 2 3 4 5 6))

;; Pulls out all the leaves of the tree, returns as a flat list.
(define (enumerate-tree tree)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (list tree))
        (else (append (enumerate-tree (car tree))
                      (enumerate-tree (cdr tree))))))

(define (count-leaves t)
  (accumulate +
              0
              (map (lambda (x) 1)
                   (enumerate-tree t)))))

;; Usage
(define tree (list 1 2 3 (list 4 5 (list 6 7))))
(count-leaves tree) ;=> 7
```

If we remove the constraint (aka hint) of using map, we can do it without enumerate-tree, but we have to make a recursive call to count-leaves if the current node is another subtree, since accumulate can't descend into the subtrees.

```
(define (count-leaves-without-map t)
  (accumulate (lambda (node count-thus-far)
    (+ count-thus-far
       (cond ((null? node) 0)
             ((not (pair? node)) 1)
             (else
               (count-leaves-without-map node))))))
  0
  t))

;; Usage
(count-leaves-without-map tree) ;=> 7
```

revc

A different version of the procedure uses enumerate-tree, just like jz, but with a slight difference

```
(define (count-leaves t)
  (accumulate (lambda (x y) (+ (length x) y)) 0 (map enumerate-tree t)))
```

Rather Iffy

```
(define (count-leaves t)
  (accumulate + 0 (map (lambda (x) 1) (enumerate-tree t))))
```

L-cent

### Procedure using anonymous recursion and append.

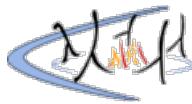
```
(define (count-leaves t)
(accumulate + 0
            (map (lambda (x) 1)
                  (((lambda (x) (x x))
                    (lambda (func)
                      (lambda (e)
                        (if (null? (cdr e))
                            (if (pair? (car e))
                                ((func func) (car e))
                                (cons (car e) '()))
                            (if (pair? (car e))
                                (append ((func func) (car e)) ((func func) (cdr e)))))))
                            (cons (car e) ((func func) (cdr e))))))) t)))
```

thanhnghuyen2187

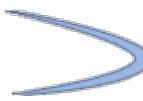
I struggled for a while, and then somehow made it work with an "almost useless" map:

```
(define (count-leaves t)
(accumulate (lambda (t accumulated)
              (+ accumulated
                 (if (pair? t)
                     (count-leaves t)
                     1)))
              0
              (map identity t)))
```

It is quite similar to jz's answer though.



# sicp-ex-2.36



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.35) | Index | Next exercise (2.37) >>

jz

Accumulate-n.

Given a sequence of sequences, applies accumulate to the first item from each, then the next item of each, etc.

Subproblem: define a proc that returns the first item from each nested sequence, and another that returns the remaining parts (accumulate-n will accumulate the former, and call itself with the latter):

```
(define (select-cars sequence)
  (map car sequence))

(define (select-cdrs sequence)
  (map cdr sequence))

;; Test
(define t (list (list 1 2 3) (list 40 50 60) (list 700 800 900)))
(select-cars t)
(select-cdrs t)

;; Accumulates the result of the first and the already-accumulated
;; rest.
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))

;; accumulate-n
(define (accumulate-n op init seqs)
  (define nil '())
  (if (null? (car seqs))
      nil
      (cons (accumulate op init (map car seqs))
            (accumulate-n op init (map cdr seqs)))))

;; Usage:
(accumulate-n + 0 t)
```

Originally I had (map (lambda (s) (car s)) sequence), but (lambda (s) (car s)) is just a function that returns car ...

Rather Iffy

But (lambda (s) (car s)) does not return car (the procedure) but the car of a pair. The notation (lambda (s) (car s)) shows the used operation in the mapping process more explicit.

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
      nil
      (cons (accumulate op init (map (lambda (s) (car s)) seqs))
            (accumulate-n op init (map (lambda (s) (cdr s)) seqs)))))

;Maybe a helpful stepping stone.
;Example in Mit-Scheme:
;(user) => (map (lambda (s) (cdr s)) '((1 2 3) (4 5 6)))
;Value: ((2 3) (5 6))
```

tf3

I would first like to set the record straight: (lambda (s) (car s)) is just a substitute for car (the procedure), it is not an expression in and of itself which is evaluated otherwise this would violate the requirement of the map function, the first argument of which has to be proc (a

procedure).

Anyway, using a map was not intuitive to me in this case while writing the solution (though it is in retrospect), hence the following solution:

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs)) '()
      (cons (accumulate op init (append-first seqs)) (accumulate-n op init (append-rest
seqs)))))

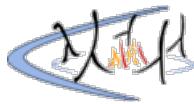
;auxiliary functions

;append the first element of each sub-list in seqs
(define (append-first seqs)
  (define (iter items res)
    (if (null? items)
        res
        (iter (cdr items) (cons (car (car items)) res))))
    (reverse (iter seqs '()))))

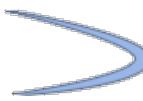
;append the cdr of each sub-list in seqs
(define (append-rest seqs)
  (define (iter items res)
    (if (null? items)
        res
        (iter (cdr items) (cons (cdr (car items)) res))))
    (reverse (iter seqs '()))))

(define (reverse items)
  (define (iter z res)
    (if (null? z)
        res
        (iter (cdr z) (cons (car z) res))))
    (iter items '())))

```



# sicp-ex-2.37



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.36) | Index | Next exercise (2.38) >>

yc

```
(define (accumulate ops initial sequence)
  (if (null? sequence)
    initial
    (ops (car sequence)
      (accumulate ops initial (cdr sequence)))))

(define (accumulate-n op init seqs)
  (if (null? (car seqs))
    ()
    (cons (accumulate op init (map car seqs))
      (accumulate-n op init (map cdr seqs)))))

(define (dot-product v w)
  (accumulate + 0 (map * v w)))

(define (matrix-*-vector m v)
  (map (lambda (w)
    (dot-product v w)) m))

(define (transpose m)
  (accumulate-n cons () m))

(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map (lambda (v) (matrix-*-vector cols v)) m)))
```

jz

I find it a bit mind-boggling that we're essentially using cons, car, cdr, and recursion/iteration to multiply matrices

```
; Accumulates the result of the first and the already-accumulated
; rest.
(define (accumulate op initial sequence)
  (if (null? sequence)
    initial
    (op (car sequence)
      (accumulate op initial (cdr sequence)))))

;; accumulate-n
(define (accumulate-n op init sequence)
  (define nil '())
  (if (null? (car sequence))
    nil
    (cons (accumulate op init (map car sequence))
      (accumulate-n op init (map cdr sequence))))))

(define matrix (list (list 1 2 3 4) (list 5 6 7 8) (list 9 10 11 12)))

(define (dot-product v1 v2)
  (accumulate + 0 (map * v1 v2)))

;; Test
(dot-product (list 1 2 3) (list 4 5 6))

;; a.
(define (matrix-*-vector m v)
  (map (lambda (m-row) (dot-product m-row v))
    m))

;; Test
(matrix-*-vector matrix (list 2 3 4 5))
```

```

;; b.
(define nil '())
(define (transpose m)
  (accumulate-n cons nil m))

;; Test
(transpose matrix)

;; c.
(define (matrix-*-matrix m n)
  (let ((n-cols (transpose n)))
    (map (lambda (m-row)
            (map (lambda (n-col)
                    (dot-product m-row n-col))
                 n-cols))
         m)))

;; But the inner map is just matrix-*-vector, so here's better:
(define (matrix-*-matrix m n)
  (let ((n-cols (transpose n)))
    (map (lambda (m-row) (matrix-*-vector n-cols m-row))
         m)))

;; Test
(matrix-*-matrix matrix (list (list 1 2) (list 1 2) (list 1 2) (list 1 2)))

```

intarga

Mine looks similar to jz's, but I'm posting with evaluation commented after the expressions, so people can double check their solutions.

```

(define my-matrix (list (list 1 2 3 4)
                        (list 4 5 6 6)
                        (list 6 7 8 9)))

(define (dot-product v w)
  (accumulate + 0 (map * v w)))

(dot-product (car my-matrix)
             (car (cdr my-matrix))) ; 56

(define (matrix-*-vector m v)
  (map (lambda (mi) (dot-product mi v)) m))

(matrix-*-vector my-matrix (car my-matrix)) ; (30 56 80)

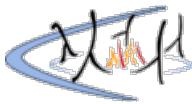
(define (transpose m)
  (accumulate-n cons '() m))

(transpose my-matrix) ; ((1 4 6) (2 5 7) (3 6 8) (4 6 9))

(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map (lambda (mi)
            (map (lambda (nj)
                    (dot-product mi nj))
                 cols))
         m)))

(matrix-*-matrix my-matrix (transpose my-matrix)) ; ((30 56 80) (56 113 161) (80 161 230))
(matrix-*-matrix (list (list 1 2)
                      (list 3 4))
                  (list (list 1 2)
                        (list 3 4))) ; ((7 10) (15 22))

```



# sicp-ex-2.38



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.37) | Index | Next exercise (2.39) >>

x davidiu

Actually, there's a much simpler way to implement `fold-left` iteratively than the book provides. Here it is:

```
(define (my-fold-left op init seq)
  (if (null? seq)
    init
    (my-fold-left op (op init (car seq)) (cdr seq))))
```

jz

```
; Accumulates the result of the first and the already-accumulated
;; rest.
(define (accumulate op initial sequence)
  (if (null? sequence)
    initial
    (op (car sequence)
        (accumulate op initial (cdr sequence)))))

(define (fold-right op initial sequence)
  (accumulate op initial sequence))

(define (fold-left op initial sequence)
  (define (iter result rest)
    (if (null? rest)
      result
      (iter (op result (car rest))
            (cdr rest))))
  (iter initial sequence))
```

```
(fold-right / 1 (list 1 2 3))
= (/ 1 (/ 2 (/ 3 1)))
= (/ 1 2/3)
= 3/2
```

```
(fold-left / 1 (list 1 2 3))
= (/ (/ (/ 1 1) 2) 3)
= (/ (/ 1 2) 3)
= (/ 1/2 3)
= 1/6
```

```
(fold-right list nil (list 1 2 3))
= (list 1 (list 2 (list 3 nil)))
= (1 (2 (3 ())))
```

I wasn't expecting that final nil to be there, but I guess it \*was\* added to the newly-created list.

```
(fold-left list nil (list 1 2 3))
= (list (list (list nil 1) 2) 3)
= (((() 1) 2) 3)
```

Same story with the innermost nil here.

Op has to be associative for fold-left and fold-right to be equivalent. For example, folding left and right with division would not be equivalent, but with matrix multiplication would (despite it's not a commutative operation).

```
(fold-left + 0 (list 1 2 3 4))
(fold-right + 0 (list 1 2 3 4))
```

The op must be commutative to ensure fold-left and fold-right get the same result. Consider sequence to be  $[x_1, x_2, \dots, x_n]$ , then (fold-left op init sequence) will be  $(op (op \dots (op init x_1) x_2) \dots x_n)$  and (fold-right op init sequence) will be  $(op (op \dots (op x_n init) x_{n-1}) \dots x_1)$ . Now consider a special case sequence only

contains one element, so sequence = [x1], then fold-left will get (op init x1) and fold-right will get (op x1 init), for these two to be equal, op must be commutative.

brother anon  
\n  
\n

examples of unfolding (brackets show priority):

1. seq = '(a), op = +, zero = 0

```
foldr: a + 0
foldl: 0 + a
```

2. seq = '(a b)

```
foldr: a + (b + 0)
foldl: (0 + a) + b
```

[foldl, foldr as given in the book]

so i'd say you need both commutativity and associativity. either alone is not enough.

hairybreeches  
\n  
\n

Suppose (foldl f a s) = (foldr f a s) for all a, s

Then (f a b) = (foldl f a (b)) = (foldr f a (b)) = (f b a)  
So f is commutative.

also  
(f a (f b c)) = (f a (f c b)) (commutativity)  
= (foldr f b (a c))  
= (foldl f b (a c))  
= (f (f b a) c)  
= (f (f a b) c) (commutativity)  
so f is associative

bxblin

Multiplication and addition properties for op works as well if we want to ensure commutative criteria.

woofy

Strict commutativity is not required. Consider OP as matrix multiplication where INITIAL value is identity marix I.  $I * A = A * I$  and also OP is associative so foldleft and foldright on matrix mulitiplication gives the same result. However matrix multiplication (the OP) is not commutative.

Update: However there is no universal Identity Matrix of all shapes! The above arguments are not valid.

Nico de Vreeze

Adding to woofy: associativity is required. Next to this either: a) the operator is commutative, as explained before. b) an identity element for the operator is given (see Monoid)

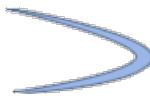
Eg:

```
(fold-left append nil (list '(1) '(2 3) '() '(4)))    ;;= (1 2 3 4)
(fold-right append nil (list '(1) '(2 3) '() '(4)))   ;;= (1 2 3 4)

;; if we use something other than nil, results will be different:
(fold-left append '(5) (list '(1) '(2 3) '() '(4)))  ;;= (5 1 2 3 4)
(fold-right append '(5) (list '(1) '(2 3) '() '(4)))  ;;= (1 2 3 4 5)
```



# sicp-ex-2.39



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.38) | Index | Next exercise (2.40) >>

jz

```
(define (fold-right op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
           (fold-right op initial (cdr sequence)))))

(define (fold-left op initial sequence)
  (define (iter result rest)
    (if (null? rest)
        result
        (iter (op result (car rest))
              (cdr rest))))
  (iter initial sequence))
```

For fold-right: use append to ensure the list is built with correct nesting, and put the first item in a list so it can be appended.

For fold-left: The inner iter in fold-left keeps peeling off the first item and doing something with it and the current result, then passing that result and the remaining items to itself. So, items are getting handled in order, from left to right, and just need to be added to the result.

```
(define (reverse-using-right items)
  (fold-right (lambda (first already-reversed)
               (append already-reversed (list first)))
             nil
             items))

(define (reverse-using-left items)
  (fold-left (lambda (result first) (cons first result))
            nil
            items))

;; Test
(define items (list 1 2 3 4 5))
(reverse-using-right items)
(reverse-using-left items)
```

Rather Iffy

```
(define (reverse-using-left items)
  (fold-left (lambda (result first) (cons first result))
            nil
            items))

;; I think it is better to make no assumptions concerning use when defining the procedure
;; that later will be passed as an argument to the formal parameter op of fold left.
;; So i prefer the use of the neutral names x and y.

(define (reverse-using-left seq)
  (fold-left (lambda (x y) (cons y x))
            '()
            seq))

;; Test
(reverse-using-left '(1 2 3 4 5))
;Value: (5 4 3 2 1)
```

Liskov

You could also define the reverse procedure using fold-right like that

```
(define (reverse-using-right sequence)
  (fold-right
    (lambda (x y)
      (fold-right cons (list x) y))
    nil
    sequence))

;Test
(reverse-using-right '(1 2 3 4 5))
;Value: (5 4 3 2 1)
```

Maybe give a name to the inner lambda function could be a good idea:

```
(define (push value sequence)
  (fold-right cons (list value) sequence))

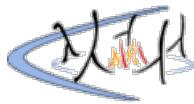
;starting from the last element, pushes all the elements on an empty list
(define (reverse-using-right sequence)
  (fold-right push nil sequence))

;Test
(reverse-using-right '(1 2 3 4 5))
;Value: (5 4 3 2 1)
```

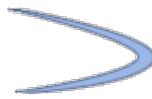
yc

```
(define (reverse sequence)
  (fold-right (lambda (x y) (append y (list x))) () sequence))

(define (reverse sequence)
  (fold-left (lambda (x y) (append (list y) x)) () sequence))
```



# sicp-ex-2.40



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.39) | Index | Next exercise (2.41) >>

```
jz

;; Supporting functions:

(define nil '())

(define (filter predicate sequence)
  (cond ((null? sequence) nil)
        ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate (cdr sequence))))
        (else (filter predicate (cdr sequence)))))

(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))

(define (enumerate-interval low high)
  (if (> low high)
      nil
      (cons low (enumerate-interval (+ low 1) high)))))

(define (flatmap proc seq)
  (accumulate append nil (map proc seq)))

(define (prime? x)
  (define (test divisor)
    (cond ((> (* divisor divisor) x) true)
          ((= 0 (remainder x divisor)) false)
          (else (test (+ divisor 1)))))
  (test 2))

(define (prime-sum? pair)
  (prime? (+ (car pair) (cadr pair)))))

(define (make-sum-pair pair)
  (list (car pair) (cadr pair) (+ (car pair) (cadr pair)))))

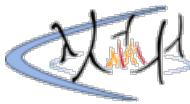
;;
-----

;; The answer ... it's just the top of page 123, pulled into a new
;; function (with flatmap):
(define (unique-pairs n)
  (flatmap (lambda (i)
             (map (lambda (j) (list i j))
                  (enumerate-interval 1 (- i 1))))
          (enumerate-interval 1 n)))

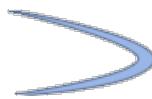
;; Test:
(unique-pairs 5)

(define (prime-sum-pairs n)
  (map make-sum-pair
       (filter prime-sum? (unique-pairs n)))))

;; Test:
(prime-sum-pairs 6)
;; => ((2 1 3) (3 2 5) (4 1 5) (4 3 7) (5 2 7) (6 1 7) (6 5 11))
```



# sicp-ex-2.41



[\[Top Page\]](#) [\[Recent Changes\]](#) [\[All Pages\]](#) [\[Settings\]](#) [\[Categories\]](#) [\[Wiki Howto\]](#)  
[\[Edit\]](#) [\[Edit History\]](#)  
 Search:

[\*\*<< Previous exercise \(2.40\)\*\*](#) | [\*\*Index\*\*](#) | [\*\*Next exercise \(2.42\) >>\*\*](#)

Woofy

My approach to generalizing to k tuples. Need thinking a bit about the leaf cases.

```

; k-tuples of [1..n]
(define (unique-tuples n k)
  (cond ((< n k) nil)
        ((= k 0) (list nil))
        (else (append (unique-tuples (- n 1) k)
                      (map (lambda (tuple) (cons n tuple))
                            (unique-tuples (- n 1) (- k 1)))))))

; application to the case of 3-tuples
(define (triples-of-sum s n)
  (filter (lambda (seq) (= (accumulate + 0 seq) s))
          (unique-tuples n 3)))
(triples-of-sum 20 30)

```

Alternative approach. More close to the exercise's style.

```

(define (unique-tuples n k)
  (define (iter m k)
    (if (= k 0)
        (list nil)
        (flatmap (lambda (j)
                   (map (lambda (tuple) (cons j tuple))
                        (iter (+ j 1) (- k 1))))
                  (enumerate-interval m n))))
    (iter 1 k)))

```

Wow, that was a very nice solution, even if it went far above and beyond the scope of the exercise in the book. Here is code for anyone here searching for a more concise solution:

```

(define (ordered-triples-sum n s)
  (filter (lambda (list) (= (accumulate + 0 list) s))
          (flatmap
            (lambda (i)
              (flatmap (lambda (j)
                         (map (lambda (k) (list i j k))
                              (enumerate-interval 1 (- j 1))))
                          (enumerate-interval 1 (- i 1))))
              (enumerate-interval 1 n))))

```

If you are having a hard time understanding this problem, study the above implementation of build-tuples.

HannibalZ

A short note to the answer before: DO NOT USE LIST as your variable name. It is an awful habit. In this case, you cannot use (list ...) since list is your variable. ps. you never how long I debug a scheme program when I use list to name one variable... ps2. sorry for inserting these words here, but I think everyone should be cautious about this.

eivanov

This solution is  $O(n^2)$  and a little optimized (doesn't check cases, when there are no triples for given i). For  $n=s=90$  runs 10 times faster, than the one on the top of the page.

```

(define (find-ordered-triples-sum n s)
  (define (k-is-distinkt-in-triple? triple)
    (let ((i (car triple))
          (j (cadr triple))
          (k (car (cdr (cdr triple)))))

      (and (> k j) (> k i))))
    (filter k-is-distinkt-in-triple?

```

```

(flatmap
  (lambda (i)
    (map (lambda (j) (list i j (- n (+ i j) ) ))
      ; j + k = n - i and k and j >= 1
      (enumerate (+ i 1) (- n (+ i 1)) )))
    (enumerate 1 (- n 2)))) ; n - 2: j and k at least 1 (i_max + 1 + 1 = n)

```

jz

```

;; Supporting functions:

(define nil '())

(define (filter predicate sequence)
  (cond ((null? sequence) nil)
        ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate (cdr sequence)))))
        (else (filter predicate (cdr sequence)))))

(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))

(define (enumerate-interval low high)
  (if (> low high)
      nil
      (cons low (enumerate-interval (+ low 1) high)))))

(define (flatmap proc seq)
  (accumulate append nil (map proc seq)))

;; Here's the unique-pairs from the chapter:
(define (unique-pairs n)
  (flatmap (lambda (i)
             (map (lambda (j) (list i j))
                  (enumerate-interval 1 (- i 1))))
    (enumerate-interval 1 n)))

;; We need to make triples (i j k). The following will do:

(define (unique-triples n)
  (flatmap (lambda (i)
             (flatmap (lambda (j)
                         (map (lambda (k) (list i j k))
                              (enumerate-interval 1 (- j 1))))
                   (enumerate-interval 1 (- i 1))))
    (enumerate-interval 1 n)))

```

Having solved this in an ugly way (without using flatmap), I got to thinking about how to generate tuples of arbitrary size (note that I started thinking about this before I clued into using flatmap, and the ugly function that appeared seemed to suggest a more general approach was required ... so some time wasted). I wanted to be able to say something like (make-tuple size max-number). A quick solution (I hope): the lists could be added by adding a number onto a list of arbitrary length, where each number is less than the number to its right (new numbers are added to the front of the list). By calling this function recursively, lists of arbitrary length can be created.

```

;; Code to build tuples of arbitrary length. Strays from the problem
;; stated in the text!

;; Example: (add-num-less-than-first (list 3 4)) will cons the numbers
;; 1 and 2 to two new copies of (list 3 4), giving (list 1 3 4) and
;; (list 2 3 4). Note that 1 and 2 are less than 3. Similarly,
;; calling this with (list 7 25) would give 6 new lists, 1 through 6
;; appended to copies of (list 7 25).
(define (add-num-less-than-first tuple)
  (map (lambda (x) (cons x tuple))
    (enumerate-interval 1 (- (car tuple) 1)))))

;; We need to have the source tuples seeded with at least one number,
;; so we'll assume those are present.
(define (build-tuples seed-tuples count-items-left-to-add)
  (cond ((= count-items-left-to-add 0) seed-tuples)
        (else
          (flatmap (lambda (t)

```

```

        (build-tuples (add-num-less-than-first t)
                      (- count-items-left-to-add 1)))
      seed-tuples)))))

;; Building the seed lists through a map, and we have 1 fewer item
;; to add to each:
(define (unique-tuples max-integer item-count)
  (build-tuples (map list (enumerate-interval 1 max-integer))
                (- item-count 1)))

;; Test:
(unique-tuples 7 3)
(unique-tuples 3 7)    ;; empty set, as expected
(unique-tuples 5 5)    ;; ((1 2 3 4 5))

```

Solving this took me many hours (an embarrassing number), with lots of false starts. I still think there's a better way to do it, but this will do for now. Any other ideas?

Now that we have this, the rest is easy. If the permutations are required, we can just use the function given in the chapter.

```

(define (unique-sets-summing-to-k set-size max-number k)
  (filter (lambda (tuple) (= (accumulate + 0 tuple) k))
          (make-unique-tuples max-number set-size)))

;; Test:
(unique-sets-summing-to-k 5 10 21)
(unique-sets-summing-to-k 3 7 10)

```

Note that this solution is generalized, which a good salaried programmer shouldn't do (unnecessary generalization = happy programmer + late project). However, for my own pain, I decided to do it.

mueen

I haven't looked at the first solution in detail.

The second solution (right above this comment) is the cleanest and simplest, but has  $O(n^3)$  complexity, when the solution can be brought down to  $O(n^2)$  by making use of the constraint that they must all add up to s.

My solution:

```

;; First get all pairs of distinct integers (i,j) that add up to s -
;; without permuting (i < j)
(define (make-pairs n s)
  (map ; Create the list of pairs
       (lambda (x) (list (- s x) x))
     (filter ; Keep only values such that the other term in the pair is
            ; smaller than x.
       (lambda (x) (and (< (- s x) x) (> (- s x) 0)))
     (enumerate-interval 2 n)))))

;; Now do it for triples, making use of the above function.
(define (make-triple n s)
  (flatmap ; Create all permutations.
    permutations
    (flatmap ; Create the triples by making use of make-pairs.
      (lambda (k)
        (map ; Convert each pair to a triple.
          (lambda (x) (cons k x))
          (make-pairs (- k 1) (- s k))))
      (filter ; Keep only values such that the other terms will be less than x.
        (lambda (x) (<= (- s x) (- (* 2 x) 3)))
        (enumerate-interval 3 n))))
  (make-triple 10 15))

```

dontbr

Apologies if there are any similarities between this solution and the others; I've just glanced at the ones above!

```

;;; Limiting ourselves to this specific exercise.

;; In accordance to the code reutilization spirit of the book,
;; this solution builds on the previous exercise (2.40), given

```

```

;; below for reference.
(define (unique-pairs n)
  (flatmap (lambda (i)
    (map (lambda (j)
      (list i j))
        (enumerate-interval 1 (- i 1))))
      (enumerate-interval 1 n)))

;; Let a (k,n)-tuple be an ordered n-tuple whose elements are at
;; most k. Then, we build (k,3)-tuples by concatenating
;; p = 1,2,3,...,k with (p-1,2)-tuples.
(define (unique-triples n)
  (flatmap (lambda (k)
    (map (lambda (pair)
      (cons k pair))
        (unique-pairs (- k 1))))
      (enumerate-interval 1 n)))

;; Finally, we run the tuplelist above through a simple filter
;; that accumulates/sums the elements of a given tuple and ensures
;; the resulting value is the one passed to the procedure.
(define (unique-triples-that-sum-to value max-element)
  (filter (lambda (triple)
    (= value (accumulate + 0 triple)))
    (unique-triples max-element)))

;; Examples:
(unique-triples 4)      ;=> ((3 2 1) (4 2 1) (4 3 1) (4 3 2))
(unique-triples-that-sum-to 7 4) ;=> ((4 2 1))

;; This can readily be generalized to (k,n)-tuples for any
;; given n.

;; Here we create (k,n)-tuples through the following recursion:
;;
;; 1) We define a (k,1)-tuple to be (k). Thus, if 'order' = 1,
;;    we return the tuplelist ((1), (2), ..., (k));
;; 2) Otherwise, we build (k,n)-tuples by concatenating
;;    p = 1,2,...k with (p-1,n-1)-tuples.
(define (unique-tuples order max-element)
  (if (= order 1)
    (map (lambda (x)
      (list x))
        (enumerate-interval 1 max-element))
    (flatmap (lambda (first)
      (map (lambda (rest)
        (cons first rest))
          (unique-tuples (- order 1) (- first 1))))
        (enumerate-interval 1 max-element)))))

;; Finally, we define a filter analogously to the one above.
(define (unique-tuples-that-sum-to value tuple-order max-element)
  (filter (lambda (tuple)
    (= value (accumulate + 0 tuple)))
    (unique-tuples tuple-order max-element)))

;; Examples:
(unique-tuples 6 7) ;=> ((6 5 4 3 2 1) (7 5 4 3 2 1) ...)
(unique-tuples-that-sum-to 22 6 7) ;=> ((7 5 4 3 2 1))

(unique-tuples 3 4)      ;=> ((3 2 1) (4 2 1) (4 3 1) (4 3 2))
(unique-tuples-that-sum-to 7 3 4) ;=> ((4 2 1))

```

atrika

```

(define (perm n seq)
  (if (= 0 n)
    (list '())
    (flatmap (lambda (elem)
      (map (lambda (next)
        (cons elem next))
          (perm (- n 1) (remove elem seq)))))
        seq)))

(define (ordered-trips-that-sum n s)
  (filter (lambda (x) (= s (accumulate + 0 x)))
    (perm 3 (enumerate-interval 1 n))))

```

kuan dontbr?, I think your unique-tuples can be simplified as below code.

```
(define (unique-tuple size max-number)
  (if (= size 0)
      (list '())
      (flatmap (lambda (i)
                  (map (lambda (t) (cons i t))
                       (unique-tuple (- size 1) (- i 1))))
              (enumerate-interval 1 max-number))))
```

Adam

Adds sum to the triple (doesn't do a separate accumulation).

```
(define (find-triplets-aggregate n s)

  (define (equal? triplet)
    (= (cadr triplet) s))

  (define (build-triplet-with-sum i j k)
    (list (list i j k) (+ i j k)))

  (flatmap (lambda (k)
              (flatmap (lambda (j)
                          (filter equal?
                                (map (lambda (i) (build-triplet-with-sum i j k))
                                     (enumerate-interval 1 (1- j)))))
                      (enumerate-interval 1 (1- k))))
            (enumerate-interval 1 n)))
```

Poc

Solution similar to Kuan's to generate tuples.

```
(define (tuples nb max-int)
  (define (iter size)
    (if (= size nb) (map list (enumerate-interval nb max-int))
        (flatmap (lambda (i)
                    (map (lambda (j) (cons j i)) (enumerate-interval size (- (car i) 1))))
                (iter (+ size 1)))))
  (iter 1))
```

vforvoid

I decided to try to get away without filtering at all, more in line with an example of generating permutations. In order to do so, we need a function that generates an ordered (by descending) list of unique positive numbers less than or equal to M that sum to a given S. We just need to generate an enumeration of all the plausible first numbers of a list and flatmap it to the recursive call result for a (length - 1) and (sum - first-item). Thus we can only generate those lists that fits task criteria and don't need a filter at all. (Written in DrRacket)

```
#lang Scheme

; ↓↓↓↓↓ Functions from the book ↓↓↓↓↓

(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))

(define (enumerate-interval low high)
  (if (> low high)
      null
      (cons low (enumerate-interval (+ low 1) high)))))

(define (flatmap proc seq)
  (accumulate append null (map proc seq)))

(define (decrease x) (- x 1))

; ↑↑↑↑↑ Functions from the book ↑↑↑↑↑

; ↓↓↓↓↓ SOLUTION ↓↓↓↓↓

(define (get-triplets-that-sum-equals-to sum max-value)
  ; Gets the sum of values of a list (n n-1 n-2 ... 2 1) of a given length
  (define (sum-enum len)
```

```

        (accumulate + 0 (enumerate-interval 1 len)))

; If we want to create an ordered (by descending) list of unique positive integers,
; we need to make sure it's starting with at least the value of its length.
; I.e. if we need a list of three items, first value should be at least 3: (3 2 1).
; If we start with 2, we will paint ourselves into a corner: (2 1 ehm 0 isn't positive)
;
; We would also need to make sure our values add up to desired sum. If we start too low
; and all the further numbers in our list should be even lower, we can't guarantee we
will
; ever amass the required sum in the end of a list. E.g. if we need to make a list of 3
; values that sum up to 10, we need to start at least with 5, because if we start with 4
; the biggest list we can get is (4 3 2) which sums to 9.
; Therefore, for a list of 3 values we can find the minimum value `x` of a first item
; like this:  $x + (x - 1) + (x - 2) \geq \text{sum}$ , e.g.  $3x - \text{sum-enum}(2) \geq \text{sum}$ . Generally, for
; a list of a length L it should be:
;  $Lx - \text{sum-enum}(L - 1) \geq \text{sum}$ 
;  $x \geq (\text{sum} + \text{sum-enum}(L - 1)) / L$ 
; In this function we take whatever is biggest: length or x
(define (get-lower-bound-for-interval len sum)
  (define m (ceiling (/ (+ sum (sum-enum (decrease len))) len)))
  (if (> m len) m len))

; Top value for an interval can be more than max-value by definition of a task.
; It also can't be higher than ( $\text{sum} - \text{sum-enum}(L - 1)$ ) where L is the length of a
current
; sequence. E.g. if we want to create an ordered (by descending) list of unique positive
; integers that sums up to 10 and we start with 7, we can only follow up with 2 and 1,
as
;  $7 + 2 + 1 = 10$ . If we start with 8, we can't create such a list.
;  $x + \text{sum-enum}(L - 1) \leq \text{sum}$ 
;  $x \leq \text{sum} - \text{sum-enum}(L - 1)$ 
(define (get-upper-bound-for-interval len sum max-value)
  (define m (- sum (sum-enum (decrease len))))
  (if (< m max-value) m max-value))

; Gets new max value for a next item of a list
(define (get-new-max-value first-item max-value)
  (define decreased-first-item (decrease first-item))
  (if (< decreased-first-item max-value)
      decreased-first-item
      max-value))

; Slight generalization of original task. Finds an ordered (by descending) list of
`length`
; distinct positive integers less than or equal to a `max-value` that sum to a given
integers.
(define (get-list-that-sum-equals-to sum len max-value)
  (if (= len 0)
      (list null)
      (flatmap (lambda (first-item)
                  (map (lambda (next-items) (cons first-item next-items))
                       (get-list-that-sum-equals-to (- sum first-item)
                                                   (decrease len)
                                                   (get-new-max-value first-item max-
value))))
              (enumerate-interval (get-lower-bound-for-interval len sum)
                                  (get-upper-bound-for-interval len sum max-
value))))))
  (get-list-that-sum-equals-to sum 3 max-value))

(get-triplets-that-sum-equals-to 20 10)

```

master

A simple solution where all concerns are fully separated into independent stages; a)  
 Enumerate all unique triples (i,j,k) such that  $1 \leq k < j \leq n$  and b) Filter out those where  $i+j+k=s$ .  
 I used `apply` instead of `accumulate` for no particular reason:

```

(define (unique-triples n)
  (flatmap (lambda (i)
              (flatmap (lambda (j)
                          (map (lambda (k)
                                  (list i j k))
                               (enumerate-interval 1 (- j 1))))
                      (enumerate-interval 1 (- i 1))))
              (enumerate-interval 1 n)))

(define (sum-to? numbers s)
  (= (apply + numbers) s))

(define (ordered-triplets-less-than-n-that-sum-to-s n s)
  (filter (lambda (triple) (sum-to? triple s)) (unique-triples n)))

```

chessweb

After comparing master's solution above to mine below I guess I understand why flatmap is called flatmap ;-)

chessweb

Even though there are simpler solutions, here is what I came up with:

```
(define (atom? x)
  (and (not (pair? x)) (not (null? x))))

; needed in (collect-triplets lst)
(define (list-of-atoms? l)
  (cond
    ((null? l) #t)
    ((atom? (car l)) (list-of-atoms? (cdr l)))
    (else #f)))

(define (triplet-sum t)
  (+ (car t) (cadr t) (caddr t)))

(define (triplet-sum-eq-s? t s)
  (= (triplet-sum t) s))

; this is ugly
(define (collect-triplets lst)
  (cond
    ((null? lst) nil)
    ((null? (car lst)) (collect-triplets (cdr lst)))
    ((and (list-of-atoms? (car (car lst))) (null? (cdr (car lst)))) (cons (car (car lst))
      (collect-triplets (cdr lst))))
     (else (append (car lst) (collect-triplets (cdr lst))))))

(define (make-ordered-triplets-summing-to-s s n)
  (filter (lambda (t) (triplet-sum-eq-s? t s))
    (collect-triplets
      (flatmap (lambda (i)
        (map (lambda (j)
          (map (lambda (k)
            (list i j k))
              (enumerate-interval 1 (- j 1)))
            (enumerate-interval 1 (- i 1)))
            (enumerate-interval 1 n)))))))

(make-ordered-triplets-summing-to-s 20 30)
```

mashomee

The obvious solution is use the *flatmap* and *unique-pairs*, though Woofy's (*unique-tuples n k*) function is awesome, the idea is similar with "Exercise 2.32" and "Example: Counting change" in chapter1.2.2. Here is my simple solution.

```
(define (unique-triple n)
  (flatmap (lambda (pair)
    (let ((k (car pair))
      (j (cadr pair)))
      (map (lambda (i)
        (list k j i))
          (enumerate-interval 1 (- j 1)))))
    (unique-pairs n)))

(assert (unique-triple 3) '((3 2 1)))

(assert (unique-triple 4) '((3 2 1) (4 2 1) (4 3 1) (4 3 2)))

(define (make-sum-eql-below s n)
  (filter (lambda (triple)
    (= s (+ (car triple)
      (cadr triple)
      (caddr triple))))
    (unique-triple n)))

(assert (make-sum-eql-below 12 15)
  '((5 4 3) (6 4 2) (6 5 1) (7 3 2) (7 4 1) (8 3 1) (9 2 1)))
```

stg101

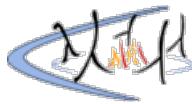
```
(define (ordered-tuples s options n-items)
```

```
(cond
  [ (and (= s 0) (= n-items 0)) (list null) ]
  [ (< s 0) null]
  [ (≤ n-items 0) null]
  [ (> n-items (length options)) null]
  [else
    (flatmap (lambda (x) (map (lambda (y) (cons x y))
                                (ordered-tuples (- s x)
                                                (remove x options)
                                                (- n-items 1))))
              options
            )
  )
)

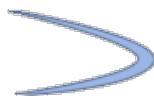
(define (ordered-triplets s n)
  (ordered-tuples s (enumerate-interval 1 n) 3)
)
```

---

Last modified : 2022-02-03 00:26:51  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-2.42



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.41) | Index | Next exercise (2.43) >>

x3v

Didn't have to use var k in the function safe?.

Not complicated once you figure out what the book's skeleton code envisions the return value of queens to be.

$((x \cdot y)^8 \cdot n)$  where n is the number of solutions.

Hope this helps.

```
(define (adjoin-position row col rest)
  (cons (list row col) rest))

(define (check a b) ; returns true if two positions are compatible
  (let ((ax (car a)) ; x-coord of pos a
        (ay (cadr a)) ; y-coord of pos a
        (bx (car b)) ; x- coord of pos b
        (by (cadr b))) ; y-coord of pos b
    (and (not (= ax bx)) (not (= ay by)) ; checks col / row
         (not (= (abs (- ax bx)) (abs (- ay by))))))) ; checks diag

(define (safe? y)
  (= 0 (accumulate + 0
                    (map (lambda (x)
                            (if (check (car y) x) 0 1))
                         (cdr y)))))

(define (queens board-size)
  (define (queen-cols k)
    (if (= k 0)
        (list '())
        (filter
         (lambda (positions) (safe? positions))
         (flatmap
          (lambda (rest-of-queens)
            (map (lambda (new-row)
                   (adjoin-position
                     new-row k rest-of-queens))
                 (enumerate-interval 1 board-size)))
          (queen-cols (- k 1))))))
    (queen-cols board-size))

  (length (queens 8)) ; 92
  (length (queens 11)) ; 2680
```

quan

Something missing from some of the other solutions is that the book specifically tells you not to check for every queen in safe?, but just for the queen in the kth column. Also, building the rows and columns and checking if they include the placed queens or not is overkill:

- same row means same (car this-queen)
- same right diagonal means same (- (car this-queen) (cadr this-queen))
- same left diagonal means same (+ (car this-queen) (cadr this-queen))

With this in mind, the solution is quite straightforward.

```
(define empty-board nil)

(define (adjoin-position new-row k rest-of-queens)
  (cons (list new-row k) rest-of-queens))

(define (queen-in-k k positions)
  (cond ((null? positions) nil)
        ((= (cadar positions) k)
         (car positions))
        (else (queen-in-k k (cdr positions)))))

(define (queens-not-k k positions)
```

```

(cond ((null? positions) nil)
      ((= (cadar positions) k)
       (cdr positions))
      (else (cons (car positions)
                  (queens-not-k k (cdr positions))))))

(define (safe? k positions)
  (let ((queen-k (queen-in-k k positions))
        (o-queens (queens-not-k k positions)))
    (null? (filter (lambda (o-q)
                      (or (= (car o-q) (car queen-k))
                          (= (- (car o-q) (cadr o-q))
                             (- (car queen-k) (cadr queen-k)))
                          (= (+ (car o-q) (cadr o-q))
                             (+ (car queen-k) (cadr queen-k))))))
                    o-queens))))
```

craig

My code may be improvable in many ways, but I'd like to submit it as it provides a solution along the lines laid down in the exercise.

Most of the sweat in this exercise comes from deciphering the code skeleton the authors provided, then working out what data structures they had in mind. That is to say, it's more of a puzzle to solve than a task to complete. I'll provide some hints based on my understanding of the exercise, followed by full code in Racket that incorporates the queens function as written in SICP.

Here's what I'm pretty sure they were driving at: the highest-level data structure is a list of board configurations. Each board configuration is a list of (x, y) coordinates locating a single queen. So the solution to (queens 1) is (((1 1)))--a single board configuration containing a single queen-coordinate. I found the word "position" really confusing, as especially in chess, "position" is often used to talk about the overall board configuration--so I used the term "coordinate" in my functions to talk about the location of an individual piece.

You will want functions to build a coordinate from a row and a column and retrieve the row or column from a coordinate, all along the lines of data structures from earlier exercises.

You might start with a dummy version of safe? that just returns #t--this will let you see that your program is building all the board combinations with one queen per column before filtering out just the ones that solve the puzzle.

When you turn your attention to safe?, what they seem to want you to do is pluck out the coordinate for the queen on a given column, and test it against each of the remaining coordinates. This can be done with an "accumulate" style function; I used the foldr that is built into Racket (and has its parameters in a different order from standard Scheme, so beware!).

```

#lang racket
(define (enumerate-interval low high)
  (cond ((> low high) null)
        ((= low high) (list high))
        (else (cons low (enumerate-interval (+ 1 low) high)))))

(define (flatmap proc seq)
  (foldr append null (map proc seq)))

(define empty-board null)

(define (safe? test-column positions)
  ;is the coordinate in the set of positions with the given column
  ;"safe" with respect to all the other coordinates (that is, does not
  ;sit on the same row or diagonal with any other coordinate)?
  ;we assume all the other coordinates are already safe with respect
  ;to each other
  (define (two-coordinate-safe? coordinate1 coordinate2)
    (let ((row1 (row coordinate1))
          (row2 (row coordinate2))
          (col1 (column coordinate1))
          (col2 (column coordinate2)))
      (if (or (= row1 row2)
              (= (abs (- row1 row2)) (abs (- col1 col2))))
          #f
          #t)))
  (let ((test-coordinate (get-coordinate-by-column test-column positions)))
    ;check the test coordinate pairwise against every other coordinate,
    ;rolling the results up with an "and," and seeding the and with
    ;an initial "true" value (because a list with one coordinate is
    ;always "safe"
    (foldr (lambda (coordinate results)
              (and (two-coordinate-safe? test-coordinate coordinate)
                   results))
          #t
          (remove test-coordinate positions))))
```

```

(define (adjoin-position new-row new-column existing-positions)
  (cons (make-coordinate new-row new-column) existing-positions))

(define (make-coordinate row column)
  (list row column))
(define (row coordinate)
  (car coordinate))
(define (column coordinate)
  (cadr coordinate))
(define (get-coordinate-by-column target-column coordinates)
  (cond ((null? coordinates) null)
        ((= target-column (column (car coordinates))) (car coordinates))
        (else (get-coordinate-by-column target-column (cdr coordinates)))))

(define (queens board-size)
  (define (queen-cols k)
    (if (= k 0)
        (list empty-board)
        (filter
          (lambda (positions) (safe? k positions))
          (flatmap
            (lambda (rest-of-queens)
              (map (lambda (new-row)
                  (adjoin-position
                    new-row k rest-of-queens))
                (enumerate-interval 1 board-size)))
            (queen-cols (- k 1))))))
  (queen-cols board-size))

```

You can use (length (queens x)) to easily count the number of solutions returned. There is one solution to (queens 1), zero to (queens 2) and (queens 3), two to (queens 4), and 92 to (queens 8).

atomik

Like the anonymous fellow below, I represented each row of the chessboard as a single number from 1 to 8. So a "board" would just be a list of 8 numbers.

This means adjoin-position can just `cons` a number onto a list and empty-board is just the empty list

```

(define (adjoin-position new-row rest-of-queens)
  (cons new-row rest-of-queens))
(define empty-board '())

```

Instead of deciding if a new row is "safe" before pushing it onto the board (as I think the authors of the SICP intended) I'm just going to push a row onto the board and then pass the whole board into `safe?`. `safe?` will then check the `car` of the board against the other rows in the board. This means our `safe?` procedure only needs to take one argument:

```

(define (safe? board)
  ; `queen` is the position of the last queen to be pushed onto the board
  ; Conveniently, it does not need to change during this procedure
  (let ((queen (car board)))
    ; As we `cdr` down our board, we need to check "down" and "diagonally"
    ; Since "down" is always the same, we can just use `queen`
    ; The right diagonal is just `(- queen 1)` and the left diagonal is
    ; `(+ queen 1)`. When we call `iter` again the right and left diagonals
    ; will be incremented and decremented.
    ; If we make it through the whole list, that means that neither our
    ; queen nor its diagonals matched a position in the board, so we return
    ; true.
    (define (iter rest-of-board right-diagonal left-diagonal)
      (cond
        ((null? rest-of-board) #t)
        ((= queen (car rest-of-board)) #f)
        ((= right-diagonal (car rest-of-board)) #f)
        ((= left-diagonal (car rest-of-board)) #f)
        (else
          (iter
            (cdr rest-of-board)
            (+ right-diagonal -1)
            (+ left-diagonal 1))))))
    ; I'm adding -1 because I don't like non-commutative operations
    (iter (cdr board) (+ queen -1) (+ queen 1))))

```

I didn't use k for functions `adjoin-position` or `safe?` and to be honest, I have no idea how I was supposed to use k in `adjoin-position`. This problem was like one of those disturbing Ikea furniture sets that have parts left over when you finish building it that just make you wonder "what was I supposed to do with that?" `k` is really handy for tracking the size of the board, though.

```

(define (queens board-size)

```

```

(define (queen-cols k)
  (if (= k 0)
      ; All of this stuff is exactly what's in the book, sans `k`
      (list empty-board)
      (filter
        (lambda (positions) (safe? positions))
        (flatmap
          (lambda (rest-of-queens)
            (map (lambda (new-row)
                    (adjoin-position new-row rest-of-queens))
                 (enumerate-interval 1 board-size)))
          (queen-cols (+ k -1))))))
  (queen-cols board-size))

```

I got up to (queens 13). There are 73,712 solutions out of  $\sim 10^{14}$  possible boards. I tried to do (queens 14) run but I got bored after a few minutes and stopped the program.

3pmtea

I'm using a representation like below:

```
((row1 col1) (row2 col2) ...) .....
```

It's a list of all solutions, where a solution is a list of coordinates, where a coordinate is (list row col).

So empty-board and adjoin-position can be defined as follows:

```

(define empty-board '())
(define (adjoin-position row col rest)
  (cons (list row col) rest))

```

The safe? procedure is a little complex:

```

(define (safe? k positions)
  (let ((trial (car positions)))
    (trial-row (caar positions))
    (trial-col (cadar positions))
    (rest (cdr positions)))
  (accumulate (lambda (pos result)
                (let ((row (car pos))
                      (col (cadr pos)))
                  (and (not (= (- trial-row trial-col)
                               (- row col)))
                       (not (= (+ trial-row trial-col)
                               (+ row col)))
                       (not (= trial-row row))
                       result)))
    true
    rest)))

```

I'm not using any procedures that are beyond the textbook, and the parameter k is not necessary here.

This method generates all 92 solutions. It will work up to (queens 9) but runs out of memory for (queens 10).

A position is represented as an ordered list 8 numbers eg (1 3 6 8 2 4 9 7 5), which denotes queen positions row 1 col 1, row 2 col 3,...,row 8 col 5

Check-row tests a row against the succeeding rows to make sure no two queens share a diagonal.

```

(define (queens n)
  (define (check-row p k)
    (define (iter k count)
      (cond ((= count (+ n 1)) true)
            (else (if (or (= (abs (- count
                                         k))
                           (abs (- (list-ref p (- count 1))
                                   (list-ref p (- k 1)))))

                           (= (+ k
                                 (list-ref p (- k 1)))
                               (+ count
                                  (list-ref p (- count 1)))))

                           false
                           (iter k (+ count 1))))))
    (iter k (+ k 1)))
  (define (check p)
    (define (iter p count)
      (cond ((= count n) true)
            (else (if (not (check-row p count))
                      false
                      (iter p (+ count 1)))))))

```

```

        (iter p (+ count 1))))))
  (filter (lambda (x) (check x))
  (permutations (enumerate-interval 1 n))))
```

```
(queens 8)
```

ctz

(just for further interest) I searched on wikipedia and it defines the "fundamental solutions" of the n queens puzzle as the solutions that cannot be converted into each other by rotation or reflection. Hence I wrote a procedure "fundamental-queens" to find the fundamental solutions. I regard a solution of the puzzle as the one sublist of the list generated by the "queens" procedure in the textbook.

```

(define (member? equiv? x lst)
;use EQUIV? to determine whether X is equivalent to some element in LST
(accumulate (lambda (x y) (or x y)) ;I wonder why I cannot directly use "or" here
#f
  (map (lambda (y) (equiv? x y))
    lst)))

(define (select-distinct equiv? lst)
;select all the distinct elements in the LST, using EQUIV? to judge whether two elements
are equivalent
(define (iter set lst)
  (cond ((null? lst) set)
    ((member? equiv? (car lst) set)
      (iter set (cdr lst)))
    (else (iter (cons (car lst) set)
      (cdr lst)))))

  (iter nil lst))

(define (queens-sol-eq? sol1 sol2)
;determine whether two solutions SOL1 and SOL2 of queens problem are equivalent
(define board-size (length sol1))
(define (reflect sol)
  (reverse-fl sol))
(define (find-col row sol)
;find which column the queen is in
(define (iter k cols)
  (if (eq? row (car cols))
    k
    (iter (inc k) (cdr cols))))
  (iter 1 sol))
(define (rot90 sol) ;rotate by 90 degrees
  (map (lambda (pos)
    (- (+ board-size 1) (find-col pos sol)))
    (enumerate-interval 1 board-size)))
(define (rot180 sol)
  (rot90 (rot90 sol)))
(define (rot270 sol)
  (rot90 (rot180 sol)))
(define (symmetry-group sol)
  (let ((ref (reflect sol)))
    (list sol (rot90 sol) (rot180 sol) (rot270 sol)
      ref (rot90 ref) (rot180 ref) (rot270 ref)))))

(member? equal? sol1 (symmetry-group sol2))

(define (fundamental-queens board-size)
  (select-distinct queens-sol-eq? (queens board-size)))

#|tests:
> (length (fundamental-queens 8))
12
> (length (queens 8))
92
> (length (fundamental-queens 6))
1
> (length (queens 6))
4
These give the same results as in Wikipedia.
|#
```

emj

Assume you have all possible queens configurations for the first k-1 columns. This is represented as a list of lists of pairs. Each list of pairs is a single queens configuration for k-1 columns that contain no checks. generate-next-col-candidates uses adjoin-position to create a candidate configuration from each of the queens configurations, with one new queen position consed to the front of each existing configuration. Note that this means for an 8 row board, the number of candidates will be 8 times the previous configurations!

Each of the candidate configurations is filtered by safe?. Safe extracts the new queen position from the

front. Then it uses checks? (see below) to look for checks with the remaining queens. Any candidate configuration that passes through the filter then becomes a safe'd configuration in the next recursion.

Like others above, I found no use in passing k to safe-candidate?

```

(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence) (accumulate op initial (cdr sequence)))))

(define (flatmap proc seq)
  (accumulate append '() (map proc seq)))

(define (enumerate-interval low high)
  (if (> low high)
      '()
      (cons low (enumerate-interval (+ 1 low) high)))))

(define (check-pos-pos? p1 p2)
  (or (same-row? p1 p2) (same-col? p1 p2) (same-diag? p1 p2)))

(define (same-diag? p1 p2)
  (= (abs (- (car p1) (car p2))) (abs (- (cdr p1) (cdr p2)))))

(define (same-col? p1 p2)
  (= (cdr p1) (cdr p2)))

(define (same-row? p1 p2)
  (= (car p1) (car p2)))

;; Add one candidate position to front of earlier
;; solution (singular) for rest-of-queens
(define (adjoin-position row col queens-config)
  (cons (cons row col) queens-config))

;; rest-of-queens is a list of list queens-configs. Each queens-config is a
;; solution for the columns checked so far.

;; To keep things simple, assume a 3 by 3 board with two queens-config and
;; last column still to add
(define rest-of-queens (list (list (cons 1 1) (cons 3 2)) (list (cons 3 1) (cons 1 2)))))

;; Note there is no 3 queen solution to t 3 by 3 board.
;; Use 3x3 for simple testing.

;; Check if a single position works with a single queens configuration:
(define (check-pos-config? pos queens-config)
  (if (null? queens-config)
      #f
      (or (check-pos-pos? pos (car queens-config)) (check-pos-config? pos (cdr queens-config)))))

  (check-pos-config? (cons 2 4) (car rest-of-queens))
  ;; #f

  ;; a candidate-config is safe if it contains no checks, hence the not
  ;; only need to check the new position in front
  (define (safe-candidate? candidate-config)
    (not (check-pos-config? (car candidate-config) (cdr candidate-config)))))

  (safe-candidate? (adjoin-position 2 4 (car rest-of-queens)))
  ;; #t
  (safe-candidate? (adjoin-position 2 4 (cadr rest-of-queens)))
  ;; #t

  ;; Test inner loop of template. Use k=3 board-size=3, as if we are adding 3rd col to a
  ;; 3x3. We just want to see that the candidate configurations are created:

  (define k 3)
  (define board-size 3)
  (flatmap
    (lambda (roqs)
      (map (lambda (new-row)
              (adjoin-position new-row k roqs))
           (enumerate-interval 1 board-size)))
    rest-of-queens)
  ;; (((1 . 3) (1 . 1) (3 . 2)) ((2 . 3) (1 . 1) (3 . 2)) ((3 . 3) (1 . 1) (3 . 2)) ((1 .
  3) (3 . 1) (1 . 2)) ((2 . 3) (3 . 1) (1 . 2)) ((3 . 3) (3 . 1) (1 . 2)))

  ;; We are close enough to go for it. Add rest of template. Like others, k is not included
  ;; in call to safe?

  (define (generate-next-col-candidates next-col board-size rest-of-queens)
    (flatmap
      (lambda (roqs)

```

```

        (map (lambda (row)
            (adjoin-position row next-col roqs))
            (enumerate-interval 1 board-size)))
        rest-of-queens)

(define board-size 3)
(generate-next-col-candidates 3 board-size rest-of-queens)
;; (((1 . 3) (1 . 1) (3 . 2)) ((2 . 3) (1 . 1) (3 . 2)) ((3 . 3) (1 . 1) (3 . 2)) ((1 . 3)
(3 . 1) (1 . 2)) ((2 . 3) (3 . 1) (1 . 2)) ((3 . 3) (3 . 1) (1 . 2)))

;; watch how candidate configs grow with recursion
(define board-size 3)
(generate-next-col-candidates 3 board-size (list (list)))
;; (((1 . 3)) ((2 . 3)) ((3 . 3)))
(generate-next-col-candidates 3 board-size (generate-next-col-candidates 3 board-size
(list (list))))
;; (((1 . 3) (1 . 3)) ((2 . 3) (1 . 3)) ((3 . 3) (1 . 3)) ((1 . 3) (2 . 3)) ((2 . 3) (2 .
3)) ((3 . 3) (2 . 3)) ((1 . 3) (3 . 3)) ((2 . 3) (3 . 3)) ((3 . 3) (3 . 3)))

(define (queens board-size)
(define (queen-cols k)
(if (= k 0)
(list (list))
(filter
(lambda (candidate) (safe-candidate? candidate))
(generate-next-col-candidates k board-size (queen-cols (- k 1)))))))
(queen-cols board-size))

(queens 4)
;; (((3 . 4) (1 . 3) (4 . 2) (2 . 1)) ((2 . 4) (4 . 3) (1 . 2) (3 . 1)))

(length (queens 8))
;; 92
;; Matches others

```

berkentekin

My solution isn't the most sophisticated one but I'll post it anyway in case it may be of help. I defined queens as (column row) unlike the book because that's closer to the chess notation.

Unfortunately the "k" parameter in safe? procedure is hanging around there doing absolutely nothing. I'm sure utilizing k would make my code much better but I just couldn't find a way to use it.

```

(define (queen col row) (list col row))
(define (col x) (car x))
(define (row x) (cadr x))

(define (contains x list)
(cond ((null? list) #f)
((equal? (car list) x) #t)
(else (contains x (cdr list)))))

(define (flatmap proc seq) (accumulate append nil (map proc seq)))

(define empty-board nil)

(define (adjoin-position k new-row rest-of-queens)
(cons (queen k new-row) rest-of-queens))

(define (safe? k positions)
(let ((newqueen (car positions))
(others (cdr positions)))
(cond
((or
(contains (row newqueen) (map row others))
(contains (- (col newqueen) (row newqueen)) (map (lambda (x) (- (col x) (row x))) others))
(contains (+ (col newqueen) (row newqueen)) (map (lambda (x) (+ (col x) (row x))) others)))
#f)
(else #t)))))

(define (queens board-size)
(define (queen-cols k)
(if (= k 0)
(list empty-board)
(filter
(lambda (positions)
(safe? k positions))
(flatmap
(lambda (rest-of-queens)
(map (lambda (new-row)
(adjoin-position

```

```

          k
          new-row
          rest-of-queens))
(enumerate-interval
 1
 board-size)))
(queen-cols (- k 1))))))
(queen-cols board-size))

```

chessweb

This solution doesn't need k in (safe? ...). With memory limit 128 MB I tested it up to (queens 12) in DrRacket where it produced 14200 solutions.

```

(define (queens board-size)
  (define empty-board '())

  (define (safe? positions)
    (define (queen-not-safe? q1 q2)
      (or (= (car q1)
              (car q2))
          (= (abs (- (car q1) (car q2)))
              (abs (- (car (cdr q1)) (car (cdr q2)))))))

    (define (iter position rest-of-queens)
      (if (null? rest-of-queens)
          #t
          (if (queen-not-safe? position (car rest-of-queens))
              #f
              (iter position (cdr rest-of-queens)))))
    (iter (car positions) (cdr positions)))

  (define (adjoin-position new-row k rest-of-queens)
    (cons (list new-row k) rest-of-queens))

  (define (queen-cols k)
    (if (= k 0)
        (list empty-board)
        (filter (lambda (positions) (safe? positions))
                (flatmap (lambda (rest-of-queens)
                           (map (lambda (new-row)
                                   (adjoin-position new-row k rest-of-queens))
                                 (enumerate-interval 1 board-size)))
                        (queen-cols (- k 1))))))
    (queen-cols board-size))

  (queens 8)
  (length (queens 8)) ; 92

```

nave

Not much novelty in terms of the solution itself, but this adds a display-board procedure which can be used to display solutions to the puzzle

```

(define nil '())

(define (enumerate-interval low high)
  (if (> low high)
      nil
      (cons low (enumerate-interval (+ low 1) high)))))

(define empty-board nil)

; adjoins a position in row and column specified by row and col
; to the existing set of positions determined by positions
; row: integer
; col: integer
; positions: list of lists which contain two integers specifying the row
; & col occupied by each position respectively
(define (adjoin-position row col positions)
  (append positions (list (list row col))))

; determines for a set of positions, whether the queen in the k-th column
; is safe with respect to the others
; how we'll tackle this:
; make one pass through positions determining which position is the k-th column position
; make another pass through the list, determining if this position is safe, defined by
; the position not being in the same row
; the position not being in the same col
; the position not being in the same diagonal (delta of row is the same as delta of col)
(define (safe? column positions)
  ; get the k-th item in a list
  ; k the number of item in the list to get

```

```

; count the count of the iterations through the list, which is always 0 to start
; items the list of items
(define (get-kth-item k count items)
  (if (= (- k 1) count)
      (car items)
      (get-kth-item k (+ count 1) (cdr items)))
(define (safe-pos? position positions)
  (if (null? positions)
      true
      (and (let ((cur-pos (car positions)))
             (or (and (= (car cur-pos) (car position)) (= (cadr cur-pos) (cadr position))) ; the position is the same as one in positions
                  (and ; the position is not...
                        (not (= (car cur-pos) (car position))) ; ...same row
                        (not (= (cadr cur-pos) (cadr position))) ; ... same col
                        (not (= (abs (- (car cur-pos) (car position)) (abs (- (cadr cur-pos) (cadr position))))))) ; ... same diagonal
                        (safe-pos? position (cdr positions))))))
          (safe-pos? (get-kth-item column 0 positions) positions)))

(define (queens board-size)
  (define (queen-cols k)
    (if (= k 0)
        (list empty-board)
        (filter
          (lambda (positions) (safe? k positions))
          (flatmap
            (lambda (rest-of-queens)
              (map (lambda (new-row)
                     (adjoin-position
                       new-row k rest-of-queens))
                  (enumerate-interval 1 board-size)))
            (queen-cols (- k 1))))))
    (queen-cols board-size))

(define (list= x y)
  (if (and (null? x) (null? y))
      true
      (and (= (length x) (length y))
           (and (= (car x) (car y)) (list= (cdr x) (cdr y))))))

(define (contains-list item sequence)
  (cond ((null? sequence) false)
        ((list= item (car sequence)) true)
        (else (contains-list item (cdr sequence)))))

(define (display-board positions board-size)
  (map (lambda (i)
         (map (lambda (j)
                (if (contains-list (list i j) positions)
                    (display "Q")
                    (display "-")))
              (enumerate-interval 1 board-size)))
       (newline))
       (enumerate-interval 1 board-size))
  (newline))

; example putting it all together with displaying full set of solutions to the Queen's
; puzzle...
; (map (lambda (solution) (display-board solution (length solution))) (queens 4))

; --Q-
; Q---
; ---Q
; -Q-- 

; ; -Q--
; ; ---Q
; ; Q---
; ; --Q-

```

# sicp-ex-2.43

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.42) | Index | Next exercise (2.44) >>

jpath

Exchanging the order of the mapping in the flatmap results in `queen-cols` being re-evaluated for every item in `(enumerate-interval 1 board-size)`. Therefore the whole work has to be duplicated `board-size` times at every recursion level. Since there are always `board-size` recursions this means that the whole work will be duplicated `board-size^board-size` times.

Therefore if the function would take  $T$  time to run for `board-size=8` with correct ordering, with the interchanged ordering it will take  $(8^8)T$  to run.

aQuaYi.com

In 2.42, every round minus 1. so it is  $(n!)$ .

In 2.43, every round is  $n$ . so it is  $(n^n)$ .

`for board-size=8, (8^8)/(8!)≈416T`

thongpv87

@aQuaYi I don't think it is correct because number of queen-cols also change over time

woofy

Not solved yet but share my thinking process:

$Q(k)$ : number of solutions of board size  $n * k$

$T(k)$ : number of steps required to calculate  $Q(k)$

original approach:  $T(k) = T(k-1) + n * Q(k-1)$

$T(0) = 1$

Louis's approach:  $T(k) = n * (T(k-1) + Q(k-1)) = n * T(k-1) + n * Q(k-1)$

$T(0) = 1$

So Louis's approach requires significantly more steps. How many more? It's tempting to say a factor of  $N^N$  due to tree recursion but it may not be the actual case.

Found this analysis for reference: <https://wernerdegroot.wordpress.com/2015/08/01/sicp-exercise-2-43/>

Alyssa P. Hacker

So, I arrived at the same result as @woofy and in the end I got these results:

Time complexity of Ex2.42 queens is  $n^4$  Time complexity of Louis Reasoner's queens is  $n^n$

So, roughly for Louis's approach time taken will be  $n^{(n-4)}T$  for  $n>4$  and equal to  $T$  for  $n\leq 4$ . These results are confirmed by actually calculating execution time of both procedures and comparing them.

nave

Given  $n$  is input size and  $k$  is  $k$  as defined in recursive iterations of

`(queen-cols (- k 1))`

original solution:

$T(k) = T(k-1)*n, T(0) = 1$

$T(n) = n*T(n-1) + n*T(n-2) + \dots + 1$

$T(n) = n!n$

Louis's solution:

$$\tau(k) = \tau(k-1)^n, \tau(0)=1$$

$$\tau(k) = \tau(k-1)^n + \tau(n-2)^n + \dots + 1$$

$$\tau(k) = n^n$$

Comparing these solutions:

$$\tau/T, \tau=n^n, T=n!n$$

$$\tau/T = n^n/n!n$$

$$\tau/T = n^{n-1}/n! \approx n^n$$

So, Louis's solution is  $n^{n-1}/n!$  times less efficient than the original which is exponentially worse on the order of  $n^n$

Kaihao

Find the link given by @woofy helpful:

<https://wernerdegroot.wordpress.com/2015/08/01/sicp-exercise-2-43/>



# sicp-ex-2.44



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.42) | Index | Next exercise (2.45) >>

iz

I am really uncomfortable writing code that I can't run ... it would be nice if the authors could have come up with an example that didn't assume the presence of other code or libraries. As it is, I'm having a bit of trouble figuring out how this works.

```
(define (up-split painter n)
  (cond ((= n 0) painter)
        (else
          (let ((smaller (up-split painter (- n 1))))
            (below painter (beside smaller))))))
```

dudrenov

The gimp comes with a scheme interpreter. Perhaps you can try it there. For me print was good enough.

bxblin

Hello,

If you are using DrRacket, follow these steps:

- 1) Install the package sicp.plt (Go to file>Install Package, type 'sicp.plt' in package source)
- 2) Paste this code '(require (planet "sicp.ss" ("soegaard" "sicp.plt" 2 1)))' in your rkt file.
- 3) Test the file with the code '(paint einstein)'. You should see a picture of Einstein in your command line.

If you want to test your code, do this:

```
(require (planet "sicp.ss" ("soegaard" "sicp.plt" 2 1)))

(define (right-split painter n)
  (if (= n 0)
    painter
    (let ((smaller (right-split painter (- n 1))))
      (beside painter (below smaller smaller)))))

(paint (right-split einstein 3))

(define (up-split painter n)
  (if (= n 0)
    painter
    (let ((smaller (up-split painter (- n 1))))
      (below painter (beside smaller smaller)))))

(paint (up-split einstein 3))
```

aQuaYi.com

Now is 2019-07-23 09:29:32

Here is Racket-lang code.

```
#lang sicp

(#%require sicp-pict)

(define (up-split painter n)
  (if (= n 0)
    painter
    (let ((smaller (up-split painter (- n 1))))
      (below painter (beside smaller smaller)))))

(paint (up-split einstein 4))
```

Nico de Vreeze

2019-12-30: Using Racket 7.5. First install sicp using file/package manager and 'sicp'. Having trouble using (load), so using #lang racket and (require)

```
#lang racket
;; file: par2.2.4.scm

(provide (all-defined-out))
```

```

(require sicp-pict)

;; (paint einstein)

;; use zorro instead of wave
(define wave mark-of-zorro)

(define wave2 (beside wave (flip-vert wave)))
(define wave4 (below wave2 wave2))

(define (flipped-pairs painter)
  (let ((painter2 (beside painter (flip-vert painter))))
    (below painter2 painter2)))

;; cannot redefine procedures
(define wave4a (flipped-pairs wave))

(define (right-split painter n)
  (if (= n 0)
      painter
      (let ((smaller (right-split painter (- n 1))))
        (beside painter (below smaller smaller)))))

;; up-split also here.
(define (up-split painter n)
  (if (= n 0)
      painter
      (let ((smaller (up-split painter (- n 1))))
        (below painter (beside smaller smaller)))))

(define (corner-split painter n)
  (if (= n 0)
      painter
      (let ((up (up-split painter (- n 1)))
            (right (right-split painter (- n 1))))
        (let ((top-left (beside up up))
              (bottom-right (below right right))
              (corner (corner-split painter (- n 1))))
          (beside (below painter top-left)
                  (below bottom-right corner)))))))

(define (square-limit painter n)
  (let ((quarter (corner-split painter n)))
    (let ((half (beside (flip-horiz quarter) quarter)))
      (below (flip-vert half) half))))

```

```

#lang racket
;; file: ex2.44.scm

(require sicp-pict)

(require "par2.2.4.scm")

(define (up-split painter n)
  (if (= n 0)
      painter
      (let ((smaller (up-split painter (- n 1))))
        (below painter (beside smaller smaller)))))

;; (paint (up-split einstein 3))
;; (paint (corner-split einstein 2))

```

If you are having trouble getting the picture language to work, I recommend using Dr. Racket and using the following code as an example (<https://gist.github.com/etscrivner/e0105d9f608b00943a49/raw/683a699aba4984998477adf0c94cd17cdfef0e3c/language.rkt>). I am now able to test the code in this section with nicely displayed pictures.

Evan

mashomee

If you are using guile with emacs geiser, **IDON'T** recommend this **guile-picture-language**.

Eventually I managed do the png painting with **Guile-Cairo**, it's rather simple to draw an png with cairo.

```

(define-module (chapter2_2_4)
  #:use-module (assert)
  #:use-module (cairo)
  #:use-module (srfi-9)           ;for record defination
  #:use-module (srfi srfi-9 gnu)   ;for record printer
  #:use-module (exercise2_38)
  #:use-module ((chapter2_2_3)
    #:select (flatmap))
  #:export (make-canvas))

(define-record-type <canvas-printer>
  (make-canvas-printer file)
  canvas-printer?
  (file canvas-print-file))

(set-record-type-printer!

```

```

<canvas-printer>
(lambda (record port)
  (format port "#<Image: ~a>" (canvas-print-file record)))

(define (make-canvas width height)
  (let* ((width width)
         (height height)
         (surface (cairo-image-surface-create 'argb32 width height))
         (cr (cairo-create surface)))
    (define (set-default)
      ;; background
      ;; (cairo-save cr)
      ;; (cairo-set-source-rgb cr 1 1 1)
      ;; (cairo-paint cr)
      ;; (cairo-restore cr)
      ;; default cairo context setting
      (cairo-set-source-rgb cr 0 0 0)
      (cairo-select-font-face cr "Sans" 'normal 'normal)
      (cairo-set-font-size cr 14.0)
      (cairo-set-line-width cr 1.0))
    (define (destroy)
      (cairo-destroy cr)
      (cairo-surface-destroy surface))
    ;; emacs caches image, so we use a new name everytime
    ;; the image changes.
    (define (png-name)
      (string-append "/tmp/geiser-"
                    (number->string (random 10000))
                    "_."
                    (number->string (random 10000))
                    ".png"))
    (define (save)
      (let ((filename (png-name)))
        (cairo-surface-write-to-png surface filename)
        (make-canvas-printer filename)))
    (set-default)
    (lambda* (cmd
              #:optional x y
              #:key file (w width) (h height) text)
      (case cmd
        ((reset)
         (destroy)
         (when w (set! width w))
         (when h (set! height h))
         (set! surface (cairo-image-surface-create 'argb32 w h))
         (set! cr (cairo-create surface))
         (set-default)
         (save))
        ((save)
         (cairo-surface-write-to-png surface file)
         (make-canvas-printer file))
        ((destroy)
         (destroy))
        ((move-to)
         (cairo-move-to cr x y))
        ((line-to)
         (cairo-line-to cr x y)
         (cairo-stroke cr))
        ((text)
         (cairo-show-text cr text)
         (save))
        ((width)
         width)
        ((height)
         height)
        ((printer)
         (save))
        (else (error "unsupported cmd:" cmd))))
    )))

(define (canvas->frame canvas)
  (make-frame (make-vector 0 0)
              (make-vector (canvas 'width) 0)
              (make-vector 0 (canvas 'height)))
  canvas)

(define (draw-line sg eg canvas)
  (canvas 'move-to (xcor-vector sg) (ycor-vector sg))
  (canvas 'line-to (xcor-vector eg) (ycor-vector eg)))

(define (segments->painter segs)
  (lambda (frame)
    (for-each
     (lambda (seg)
       (draw-line
        ((frame-coord-map frame) (start-segment seg))
        ((frame-coord-map frame) (end-segment seg))
        (canvas-frame frame)))
     segs)
    (canvas-frame frame) 'printer)))

(define wave-lt '((0 . 28) (29 . 73) (56 . 64) (74 . 64) (64 . 28) (69 . 15) (74 . 0)))
(define wave-rt '((-110 . 0) (119 . 28) (110 . 64) (137 . 64) (182 . 118)))
(define wave-rb '((-137 . 182) (109 . 99) (182 . 154)))
(define wave-cb '((-110 . 182) (92 . 127) (74 . 182)))
(define wave-lb '((-46 . 182) (65 . 91) (55 . 73) (29 . 109) (0 . 64)))

;; ((110 . 182) (92 . 127))
(define (shrink-to-lxl seg base)
  (let ((sg (start-segment seg)))

```

```

        (eg (end-segment seg)))
(list (cons (exact->inexact (/ (xcor-vector sg) base))
            (exact->inexact (/ (ycor-vector sg) base)))
        (cons (exact->inexact (/ (xcor-vector eg) base))
            (exact->inexact (/ (ycor-vector eg) base))))))
(assert (shrink-to-lx1 '((110 . 182) (92 . 127)) 182)
      '((0.6043956043956044 . 1.0) (0.5054945054945055 . 0.6978021978021978)))

(define (wave-lines->segment-list lines)
  (map
    (lambda (seg)
      (shrink-to-lx1 seg 182))
    (foldr append '()
      (map
        (lambda (line)
          (foldr
            (lambda (point result)
              (cond ((null? result)
                     (list point))
                    ((and (null? (cdr result))
                          (not (list? (car result))))))
                     (list (list point (car result)))))
            (else
              (cons (list point (caar result))
                    result)))))
        '()
        line)))
  lines)))

(define (make-wave)
  (segments->painter
    (wave-lines->segment-list (list wave-cb wave-lb wave-rb wave-rt wave-lt)))))

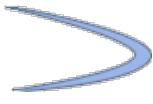
(define c (make-canvas 500 500))
(define p (make-wave))
(p (make-frame (make-vector 200 10)
               (make-vector 0 300)
               (make-vector 100 0)
               c))
(p (canvas->frame c))
(c 'printer)
(c 'destroy)

```

the picture comes from sicp online book [sicp online book](#)



# sicp-ex-2.45



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.44) | Index | Next exercise (2.46) >>

jz

```
(define (split orig-placer split-placer)
  (lambda (painter n)
    (cond ((= n 0) painter)
          (else
            (let ((smaller ((split orig-placer split-placer) painter (- n 1))))
              (orig-placer painter (split-placer smaller smaller)))))))
```

brave brother

i think explicit internal func makes it clearer. but the same sure.

```
(define (split f g)
  (define (rec painter n)
    (if (= n 1)
        painter
        (let ((smaller (rec painter (- n 1))))
          (f painter (g smaller smaller)))))

  rec)
```

woofy

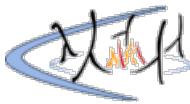
Also including left-split and down split?

```
(define (split main sub flip)
  (define (do painter n)
    (if (= n 0)
        painter
        (let ((smaller (do painter (- n 1))))
          (flip (main painter (sub smaller smaller))))))

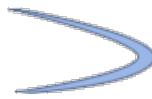
  do)

(define right-split (split beside below identity))
(define up-split (split below beside identity))
(define left-split (split beside below flip-horiz))
(define down-split (split below beside flip-vert))

; or this is better already
(define left-split (compose flip-horiz right-split))
(define down-split (compose flip-vert down-split))
```



# sicp-ex-2.46



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.45) | Index | Next exercise (2.47) >>

jz

I defined some simple tests to make sure that everything works. Obviously total overkill for the problem at hand, but I wanted to try setting up simple tests in this language like I do for all other languages I use.

```
; ; Testing:
(define (ensure b err-msg)
  (if (not b) (error err-msg)))

(define (ensure-all list-of-tests-and-messages)
  (cond ((null? list-of-tests-and-messages) true)
        (else
          (ensure (car list-of-tests-and-messages)
                  (cadr list-of-tests-and-messages))
          (ensure-all (cddr list-of-tests-and-messages)))))

;; Tests:
(define v2-3 (make-vect 2 3))
(define v5-8 (make-vect 5 8))

;; Simple TDD to ensure everything works. Yes, pretty keen of me.
(ensure-all
  (list (= (xcor-vect (make-vect 3 4)) 3) "x"
        (= (ycor-vect (make-vect 3 4)) 4) "y"
        (eq-vect? (make-vect 7 11) (add-vect v5-8 v2-3)) "add"
        (eq-vect? (make-vect 3 5) (sub-vect v5-8 v2-3)) "sub"
        (eq-vect? (make-vect 10 16) (scale-vect 2 v5-8)) "scale"
  ))

;; -----
(define (make-vect x y) (cons x y))
(define (xcor-vect vec) (car vec))
(define (ycor-vect vec) (cdr vec))

(define (eq-vect? v1 v2)
  (and (= (xcor-vect v1) (xcor-vect v2))
       (= (ycor-vect v1) (ycor-vect v2))))

(define (add-vect v1 v2)
  (make-vect (+ (xcor-vect v1) (xcor-vect v2))
             (+ (ycor-vect v1) (ycor-vect v2))))
(define (sub-vect v1 v2)
  (make-vect (- (xcor-vect v1) (xcor-vect v2))
             (- (ycor-vect v1) (ycor-vect v2))))
(define (scale-vect s vec)
  (make-vect (* s (xcor-vect vec))
             (* s (ycor-vect vec))))
```

saving a few lines of code

```
(define (f-vect v1 v2 f)
  (cons (f (xcor-vect v1)
            (xcor-vect v2))
        (f (ycor-vect v1)
            (ycor-vect v2)))))

(define (add-vect v1 v2)
  (f-vect v1 v2 +))

(define (sub-vect v1 v2)
  (f-vect v1 v2 -))
```



# sicp-ex-2.47

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.46) | Index | Next exercise (2.48) >>

jz

This question seems out of place with the others -- it's too easy.

```
; First implementation.

(define (make-frame origin edge1 edge2)
  (list origin edge1 edge2))
(define (frame-origin f) (car f))
(define (frame-edge1 f) (cadr f))
(define (frame-edge2 f) (caddr f))

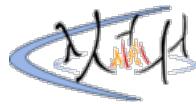
;; Test just using scalars, lazy. That's there's no type safety might
;; be worrisome to some coming from Java/C++.
(define f (make-frame 1 2 3))
(frame-origin f)
(frame-edge1 f)
(frame-edge2 f)

; Second imp.

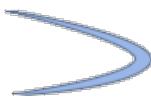
(define (make-frame origin edge1 edge2)
  (cons origin (cons edge1 edge2)))
(define (frame-origin f) (car f))
(define (frame-edge1 f) (cadr f))
(define (frame-edge2 f) (caddr f))

; same test as before applies.
```

Last modified : 2022-06-03 14:32:41  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-2.48



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.47) | Index | Next exercise (2.49) >>

dudrenov

*;; Exercise 2.48*

```
;; heh
(define make-segment cons)
(define start-segment car)
(define end-segment cdr)
```

electraRod

*; exercise says to use vector representation*

```
(define (make-segment start-point end-point)
  (list (make-vect 0 start-point)
        (make-vect 0 end-point)))
(define (start-segment seg)
  (car seg))
(define (end-segment seg)
  (cadr seg))
```

jirf

From ex2.6

*> A two-dimensinal vector v running from the origin to a point can be represented as a pair consisting of an x-coordinate and a y-coordinate.*

The above vector implementation has a constructor that takes a number and what I assume is a xy pair, which is probably a mistake. If the second parameter is a number than I suppose the constructor always produces xy pairs of the form (0,y)

Should be

```
;; Book said pair of x y but I like using lists where I can
(define (make-vect x y)
  (list x y))

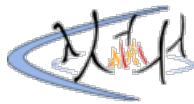
;; Constructor

(define (make-segment start end)
  (list start end))

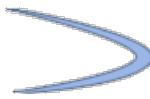
;; Selectors

(define (start-segment segment)
  (car segment))

(define (end-segment segment)
  (cadr segment))
```



# sicp-ex-2.49



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.48) | Index | Next exercise (2.50) >>

x3v

Used DrRacket instead of emacs for this chapter as there is an sicp-compatible graphics library. Imports shown below. Use the segments->painter procedure provided by the library, unless you want to implement draw-line yourself. The compatible procedures are vect and segment instead of make-vect and make-segment. Hope this helps.

```
#lang sicp
(#%require sicp-pict)

;; Exercise 2.49 - Use the segments->painter procedure provided by the library
;; Procedures to use: vect, segment; instead of make-vect and make-segment

;; Exercise 2.49a
(define outline
  (segments->painter
   (list
    (segment (vect 0.0 0.0) (vect 0.0 1.0))
    (segment (vect 0.0 0.0) (vect 1.0 0.0))
    (segment (vect 0.0 1.0) (vect 1.0 1.0))
    (segment (vect 1.0 0.0) (vect 1.0 1.0)))))

;; (paint outline)

;; Exercise 2.49b
(define x-painter
  (segments->painter
   (list
    (segment (vect 0.0 0.0) (vect 1.0 1.0))
    (segment (vect 0.0 1.0) (vect 1.0 0.0)))))

;; (paint x-painter)

;; Exercise 2.49c
(define diamond
  (segments->painter
   (list
    (segment (vect 0.0 0.5) (vect 0.5 1.0))
    (segment (vect 0.5 1.0) (vect 1.0 0.5))
    (segment (vect 1.0 0.5) (vect 0.5 0.0))
    (segment (vect 0.5 0.0) (vect 0.0 0.5)))))

;; Exercise 2.49d - Not sure if the below implementation unnecessarily complicates the
;; process
;; Measurements of frame on kindle x: 1.5 y: 1.8
(define x 1.5)
(define y 1.8)

;; Define helper functions
;; takes a list of coords with the format x1 y1 x2 y2 and normalises them wrt measurements
(define (normalize coords)
  (let ((x1 (car coords))
        (y1 (cadr coords))
        (x2 (caddr coords))
        (y2 (caddr coords)))
    (list (/ x1 x) (/ y1 y) (/ x2 x) (/ y2 y))))

;; symmetry will be useful
(define (mirror coords)
  (list (- x (car coords)) (cadr coords) (- x (caddr coords)) (caddr coords)))

;; takes a list of coords converts to a line segment
(define (list->line coords)
  (let ((x1 (car coords))
        (y1 (cadr coords))
        (x2 (caddr coords))
        (y2 (caddr coords)))
    (segment (vect x1 y1) (vect x2 y2))))

;; define wave coords
(define wave-coords
  (list
   (list 0.5 1.5 0.65 y))
```



```
(make-segment r t)))  
frame)))
```

mathieu @caesarjuly Unless '/' is overloaded somehow and performs division on pairs, your solution will not work. Also, there's no need to pass the frame argument to your painters, segments->painter returns a lambda that will accept the frame as it's argument.

brave one

```
; sorry no make-<something> and selectors here, too much to type!  
  
; a.  
(define outline  
  (let ((segments '()  
        (((0 0) (0 1))  
         ((0 1) (1 1))  
         ((1 1) (1 0))  
         ((1 0) (0 0))))  
    (segments->painter segments)))  
  
; b.  
(define cross  
  (let ((segments '()  
        (((0 0) (1 1))  
         ((0 1) (1 0))))  
    (segments->painter segments)))  
  
; c.  
(define diamond  
  (let ((segments '()  
        (((0 0.5) (0.5 1))  
         ((0.5 1) (1 0.5))  
         ((1 0.5) (0.5 0))  
         ((0.5 0) (0 0.5))))  
    (segments->painter segments)))
```

SophiaG

The wording of the exercise seems to imply these functions should take a frame as input and calculate the segments based on that, which would also be much more useful should one intend to actually use them for drawing. Here's my answer given that:

(Also including coordinates for wave, aka George, pulled from a comment on Weiqun Zhang's blog by someone calling themselves "physjam")

```
(define (outline->painter frame)  
  (let ((origin2 (make-vect  
                  (- (xcor-vect (edge2-frame frame))  
                     (xcor-vect (origin-frame frame)))  
                  (- (ycor-vect (edgel-frame frame))  
                     (ycor-vect (origin-frame frame))))))  
    (segments->painter  
     (list  
       (make-segment (origin-frame frame) (edgel-frame frame))  
       (make-segment (edgel-frame frame) origin2)  
       (make-segment origin2 (edge2-frame frame))  
       (make-segment (edge2-frame frame) (origin-frame frame))))))  
  
(define (X->painter frame)  
  (let ((origin2 (make-vect  
                  (- (xcor-vect (edge2-frame frame))  
                     (xcor-vect (origin-frame frame)))  
                  (- (ycor-vect (edgel-frame frame))  
                     (ycor-vect (origin-frame frame))))))  
    (segments->painter  
     (list  
       (make-segment (origin-frame frame) origin2)  
       (make-segment (edgel-frame frame) (edge2-frame frame))))))  
  
(define (diamond->painter frame)  
  (let ((midpoint1 (sub-vect (edgel-frame frame) (origin-frame frame)))  
        (midpoint2 (sub-vect origin2 (edgel-frame frame)))  
        (midpoint3 (sub-vect origin2 (edge2-frame frame)))  
        (midpoint4 (sub-vect (edge2-frame frame) (origin-frame frame))))  
    (segments->painter  
     (list  
       (make-segment midpoint1 midpoint2)  
       (make-segment midpoint2 midpoint3)  
       (make-segment midpoint3 midpoint4)  
       (make-segment midpoint4 midpoint1))))))
```

```

(segments->painter
(list
  (make-segment midpoint1 midpoint2)
  (make-segment midpoint2 midpoint3)
  (make-segment midpoint3 midpoint4)
  (make-segment midpoint4 midpoint1)))))

(define wave
  (segments->painter (list
    (make-segment (make-vec .25 0) (make-vec .35 .5))
    (make-segment (make-vec .35 .5) (make-vec .3 .6))
    (make-segment (make-vec .3 .6) (make-vec .15 .4))
    (make-segment (make-vec .15 .4) (make-vec 0 .65))
    (make-segment (make-vec 0 .65) (make-vec 0 .85))
    (make-segment (make-vec 0 .85) (make-vec .15 .6))
    (make-segment (make-vec .15 .6) (make-vec .3 .65))
    (make-segment (make-vec .3 .65) (make-vec .4 .65))
    (make-segment (make-vec .4 .65) (make-vec .35 .85))
    (make-segment (make-vec .35 .85) (make-vec .4 1))
    (make-segment (make-vec .4 1) (make-vec .6 1))
    (make-segment (make-vec .6 1) (make-vec .65 .85))
    (make-segment (make-vec .65 .85) (make-vec .6 .65))
    (make-segment (make-vec .6 .65) (make-vec .75 .65))
    (make-segment (make-vec .75 .65) (make-vec 1 .35))
    (make-segment (make-vec 1 .35) (make-vec 1 .15))
    (make-segment (make-vec 1 .15) (make-vec .6 .45))
    (make-segment (make-vec .6 .45) (make-vec .75 0))
    (make-segment (make-vec .75 0) (make-vec .6 0))
    (make-segment (make-vec .6 0) (make-vec .5 .3))
    (make-segment (make-vec .5 .3) (make-vec .4 0))
    (make-segment (make-vec .4 0) (make-vec .25 0))
  )))

;George!

```

Lera

Here is only d-solution

```

(define (do-many-vectors x-coords y-coords)
  (let ((coeff (/ 1 4.8)))
    (map (lambda (x y) (vector-scale coeff (make-vec x y))) x-coords y-coords)))
  ; I just measured with a ruler all distances on the screen and then divided
  by side of frame all measured distances

(define (make-many-segments start-vectors end-vectors)
  (map (lambda (start-vector end-vector) (make-segment start-vector end-vector))
    start-vectors
    end-vectors))

(define start-vectors (do-many-vectors
  (list 1.2 1.7 1.5 0.7 0 0.7 1.5 1.9 1.7 2.9 3.1 2.9 3.6 2.9
2.9 2.9 2.4)
  (list 0 2.35 2.7 1.9 4.0 2.8 3.05 3.05 3.9 4.8 3.9 3.05 3.05
2.1 2.1 0 1.4)))
  (define end-vectors (do-many-vectors
    (list 1.7 1.5 0.7 0 0.7 1.5 1.9 1.7 1.9 3.1 2.9 3.6 4.8 4.8
3.6 2.4 2.0)
    (list 2.35 2.7 1.9 3.1 2.8 3.05 3.05 3.9 4.8 3.9 3.05 3.05 1.8
0.75 0 1.4 0)))

(define list-of-wave (make-many-segments start-vectors end-vectors))

(define wave (segments->painter list-of-wave))

```

# sicp-ex-2.50

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.49) | Index | Next exercise (2.51) >>

dudrenov

*; That's one way of doing it.*

```
(define (flip-horiz painter)
  (transform-painter painter
    (make-vect 1.0 0.0)
    (make-vect 0.0 0.0)
    (make-vect 1.0 1.0)))

(define (repeated fn t)
  (if (= t 1)
    fn
    (lambda (x)
      (fn ((repeated fn (- t 1))
           x)))))

(define (rotate180 painter)
  ((repeated rotate90 2) painter))

(define (rotate270 painter)
  ((repeated rotate90 3) painter))
```

charlesdalton

*; The other way of doing it ;)*

```
(define (flip-horiz painter)
  (transform-painter painter
    (make-vect 1.0 0.0)
    (make-vect 0.0 0.0)
    (make-vect 1.0 1.0)))

(define (rotate180 painter)
  (transform-painter painter
    (make-vect 1.0 1.0)
    (make-vect 0.0 1.0)
    (make-vect 1.0 0.0)))

(define (rotate270 painter)
  (transform-painter painter
    (make-vect 0.0 1.0)
    (make-vect 0.0 0.0)
    (make-vect 1.0 1.0)))
```

ceasarxinsanum

Here is lazy method of doing it

```
(define (rotate180 painter)
  (rotate90 (rotate90 painter)))

(define (rotate270 painter)
  (rotate90 (rotate90 (rotate90 painter))))
```

# sicp-ex-2.51

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.50) | Index | Next exercise (2.52) >>

```
(define (below painter1 painter2)
  (let ((split-point (make-vect 0.0 0.5)))
    (let ((paint-bottom
           (transform-painter painter1
                           (make-vect 0.0 0.0)
                           (make-vect 1.0 0.0)
                           split-point))
          (paint-top
           (transform-painter painter2
                           split-point
                           (make-vect 1.0 0.5)
                           (make-vect 0.0 1.0))))
      (lambda (frame)
        (paint-bottom frame)
        (paint-top frame)))))

(define (below-2 painter1 painter2)
  (rotate90 (beside (rotate270 painter1) (rotate270 painter2)))))

;; Another way
(define (below-2 painter1 painter2)
  (rotate270 (beside (rotate90 painter2) (rotate90 painter1))))
```

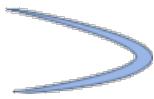
```
;; A way to do it using only the operations created in
;; ex 2.50: rotate180 and rotate270
;; It's a bit of a wild ride.

(define (below3 p1 p2)
  (rotate180 (rotate270 (beside (rotate270 p1) (rotate270 p2)))))
```

Last modified : 2021-05-13 21:38:55  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-2.52



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.51) | Index | Next exercise (2.53) >>

```
;; a
(define wave
  (segments->painter (list
    ;; ...
    (make-segment (make-vec 0.44 0.7) (make-vec 0.51 0.7)))))

;; b
(define (corner-split painter n)
  (if (= n 0)
    painter
    (beside (below painter (up-split painter (- n 1)))
            (below (right-split painter (- n 1)) (corner-split painter (- n 1))))))

;; c
(define (square-limit painter n)
  (let ((combine4 (square-of-four flip-vert rotate180
                                    identity flip-horiz)))
    (combine4 (corner-split painter n))))
```

HYC

should it be n instead of (- n 1) for up-split and right-split? Since (right-split painter 0) will return painter, which makes it results different from the original corner-split.

```
;; b
(define (corner-split painter n)
  (if (= n 0)
    painter
    (beside (below painter (up-split painter n))
            (below (right-split painter n) (corner-split painter (- n 1))))))
```

Last modified : 2019-08-23 02:22:47  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**

# sicp-ex-2.53

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

```
(list 'a 'b 'c)
;; (a b c)

(list (list 'george))
;; ((george))

(cdr '((x1 x2) (y1 y2)))
;; (y1 y2)

(cadr '((x1 x2) (y1 y2)))
;; (y1 y2)

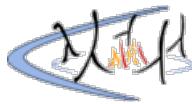
(pair? (car '(a short list)))
;; #f

(memq 'red '((red shoes) (blue socks)))
;; #f

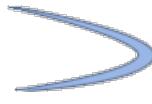
(memq 'red '(red shoes blue socks))
;; (red shoes blue socks)
```

<< Previous exercise (2.52) | Index | Next exercise (2.54) >>

Last modified : 2010-08-06 12:42:09  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-2.54



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.53) | Index | Next exercise (2.55) >>

```
(define (equal? list1 list2)
  (cond ((and (not (pair? list1)) (not (pair? list2)))
          (eq? list1 list2))
         ((and (pair? list1) (pair? list2))
          (and (equal? (car list1) (car list2)) (equal? (cdr list1) (cdr list2))))
         (else false)))

(equal? '(1 2 3 (4 5) 6) '(1 2 3 (4 5) 6))
;Value: #t

(equal? '(1 2 3 (4 5) 6) '(1 2 3 (4 5 7) 6))
;Value: #f
```

andras ; 2.54

```
;; a equal? b if, and only if:
;; either eq? a and b, or
;; a and b are both pairs or they are both nil,
;; and (car a) equal? (car b), and (cdr a) equal? (cdr b)
(define (equal? a b)
  (if (or
        (eq? a b)
        (and
         (or
          (and
           (pair? a)
           (pair? b))
          (and
           (null? a)
           (null? b)))
         (and
          (equal? (car a) (car b))
          (equal? (cdr a) (cdr b))))))
    #t #f))
```

miranda

; 2.54

;; the above solution works, but a good rule of thumb is that any time you have  
;; the pattern "if <a> then <true> else <false>", you can replace the whole  
;; thing by <a>, so andras' solution becomes:

```
(define (equal2? a b)
  (or
   (eq? a b)
   (and
    (or
     (and
      (pair? a)
      (pair? b))
     (and
      (null? a)
      (null? b)))
    (and
     (equal? (car a) (car b))
     (equal? (cdr a) (cdr b))))))

;; I also coded a version which is a little less booleany -- it seems clearer to me,
;; but perhaps that's just because I wrote it. ;)

(define (equal3? l1 l2)
  (if (and (pair? l1) (pair? l2))
      (cond ((null? l1) (null? l2))
            ((null? l2) false)
            ((equal2? (car l1) (car l2)) (equal2? (cdr l1) (cdr l2))))
```

```
(else false))  
(eq? 11 12)))
```

vlad ; 2.54

```
;whilst the above two solutions are correct, I find them rather counterintuitive  
;at first sight  
;it seems like eq? actually handles the cases where:  
; one of the operands is '() or nill, as well as only one of the operands being a pair  
; thus checking for these cases is unnecessary  
  
(define (equal2? a b)  
  (if (and (pair? a) (pair? b))  
      (and (equal2? (car a) (car b)) (equal2? (cdr a) (cdr b)))  
      (eq? a b)))
```

meteorgan

The answer is relevant to the procedure eq?. In my IDE: DrRacket 5.1. eq? can do everything described by the question 2.54, so we can use (eq? list1 list2) to solve the problem. If we just think eq? is used to check symbols. the following code works.

```
(define (equal1? x y)  
  (cond ((and (null? x) (null? y)) #t)  
        ((and (symbol? x) (symbol? y)) (eq? x y))  
        ((and (pair? x) (pair? y))  
         (equal1? (car x) (car y)) (equal1? (cdr x) (cdr y))) #t)  
        (else #f)))
```

AMS

I find the following easier to follow...

```
(define (equal? list1 list2)  
  (cond ((and (null? list1) (null? list2)) true)  
        ((or (null? list1) (null? list2)) false)  
        ((not (eq? (car list1) (car list2))) false)  
        (else (equal? (cdr list1) (cdr list2)))))
```

palatin

I thought the following would work, but it triggers an error related to use of macro name as a variable on Gambit Scheme:

```
(define (equal1? list1 list2)  
  (accumulate and #t (map eq? list1 list2)))
```

So I settled with this, which I find more concise, albeit not the most efficient:

```
(define (equal1? list1 list2)  
  (accumulate (lambda (x y) (and x y)) #t (map eq? list1 list2)))
```

Daniel-Amariei

Interesting exercise.

```
(define (equal? L1 L2)  
  (cond ((and (pair? L1)  
              (pair? L2)) (if (eq? (car L1) (car L2))  
                           (equal? (cdr L1) (cdr L2))  
                           false))  
        ((and (not (pair? L1))  
              (not (pair? L2))) (eq? L1 L2))  
        (else false)))  
  
(equal? '(this is a list) '(this is a list)) ; true  
(equal? '(this is a list) '(this (is a list))) ; false
```

atrika

this shouldn't be used to compare lists containing numbers

```
(define (my-equal? a b)
  (if (and (pair? a) (pair? b) (eq? (car a) (car b)))
      (my-equal? (cdr a) (cdr b))
      (eq? a b)))
```

fubupc

@palatin's solution is wrong.

e.g. (equal1? '(1 2 3) '(1 2 3 4)) => #t

which is obviously incorrect.

adam

I find this solution to be much easier to understand.

```
(define (equal? lat1 lat2)
  (define (lists-empty?)
    (and (null? lat1) (null? lat2)))

  (define (either-list-empty?)
    (or (null? lat1) (null? lat2)))

  (cond ((lists-empty?) #t)
        ((either-list-empty?) #f)
        ((eq? (car lat1) (car lat2)) (equal? (cdr lat1) (cdr lat2))))))
```

Isaac

Naturally, I find my solution easiest to understand

```
(define (equal? a b)
  (or (and
        (not (pair? a))
        (not (pair? b))
        (eq? a b))
      (and
        (pair? a)
        (pair? b)
        (equal? (car a) (car b))
        (equal? (cdr a) (cdr b)))))
```

tugs

Reduce more lines

```
(define (equal? a b)
  (or
    (eq? a b)
    (and
      (pair? a)
      (pair? b)
      (equal? (car a) (car b))
      (equal? (cdr a) (cdr b)))))
```

# sicp-ex-2.55

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.54) | Index | Next exercise (2.56) >>

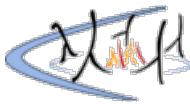
Lenatis

; ; 2.55

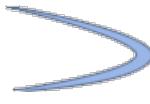
```
; ; (car ''something) is treated by the interpreter as:  
; ; (car (quote (quote something)))  
; ; The first occurrence of 'quote' quotes the next entity  
; ; (quote something), which is actually a list with two elements, so  
; ; caring this list yields 'quote. However, this is just a quoted  
; ; symbol, not a procedure, typing quote in the interpreter prints:  
  
quote  
  
; ; => (#@keyword . #<primitive-macro! #<primitive-procedure quote>>)  
; ; whereas typing 'quote just yielded it literally.
```

note that (car '(list 'a)) returns list. Similarly, "abracadabra is translated as '(quote abracadabra) and car it will return "quote" (the second quote)

Last modified : 2019-08-13 23:48:52  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-2.56



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.55) | Index | Next exercise (2.57) >>

```
(define (deriv expr var)
  (cond ((number? expr) 0)
        ((variable? expr) (if (same-variable? expr var) 1 0))
        ((sum? expr) (make-sum (deriv (addend expr) var)
                               (deriv (augend expr) var)))
        ((product? expr) (let ((m1 (multiplier expr)) (m2 (multiplicand expr)))
                           (make-sum (make-product (deriv m1 var) m2)
                                     (make-product m1 (deriv m2 var)))))
        ((and (exponentiation? expr) (=number? (deriv (exponent expr) var) 0))
              (let ((b (base expr)) (e (exponent expr)))
                (make-product (deriv b var)
                              (make-product e (make-exponentiation b (make-sum e -1))))))
        (else (list 'deriv expr var))))
```

Without the check that the derivative of the exponent is 0 the code would be wrong, since the derivative of  $x \rightarrow x^x$  is not  $x \rightarrow x^x \cdot x^{x-1} = x^x$ .

NTeGrotenhuis

Show how to extend the basic differentiator to handle more kinds of expressions. For instance, implement the differentiation rule

$$\frac{d(u^n)}{dr} = nu^{(n-1)}(du/dr)$$

by adding a new clause to the deriv program and defining appropriate procedures exponentiation?, base, exponent, and make-exponentiation. (You may use the symbol  $^{**}$  to denote exponentiation.) Build in the rules that anything raised to the power 0 is 1 and anything raised to the power 1 is the thing itself.

code given in the book:

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
           (make-product (multiplier exp)
                         (deriv (multiplicand exp) var))
           (make-product (deriv (multiplier exp) var)
                         (multiplicand exp))))
        (else
         (error "unknown expression type -- DERIV" exp))))
(define (variable? x) (symbol? x))

(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))

(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2)) (+ a1 a2))
        (else (list '+ a1 a2)))))

(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (number? m1) (number? m2)) (* m1 m2))
        (else (list '* m1 m2)))))

(define (=number? exp num)
  (and (number? exp) (= exp num)))

(define (sum? x)
  (and (pair? x) (eq? (car x) '+)))

(define (addend s) (cadr s))
```

```

(define (augend s) (caddr s))

(define (product? x)
  (and (pair? x) (eq? (car x) '*)))

(define (multiplier p) (cadr p))

(define (multiplicand p) (caddr p))

```

First add this code to (derive exp var) to add differentiation of exponentiations.

```

((exponentiation? exp)
  (make-product
    (make-product (exponent exp)
      (make-exponentiation (base exp)
        (make-sum(exponent exp) '-1)))
      (deriv (base exp) var)))

```

The end product is:

```

(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var))))
        ((product? exp)
         (make-sum
           (make-product (multiplier exp)
             (deriv (multiplicand exp) var))
           (make-product (deriv (multiplier exp) var)
             (multiplicand exp)))))
        ((exponentiation? exp)
         (make-product
           (make-product (exponent exp)
             (make-exponentiation (base exp)
               (if (number? (exponent exp))
                   (- (exponent exp) 1)
                   ('(- (exponent exp) 1))))))
           (deriv (base exp) var)))
        (else
         (error "unknown expression type -- DERIV" exp))))

```

Next (define (exponentiation? exp))

```

(define (exponentiation? exp)
  (and (pair? exp) (eq? (car exp) '**)))

```

We are using `**` as the symbol for exponent.

Next, (define (base exp)) and (define (exponent exp))

```

(define (base exp)
  (cadr exp))

(define (exponent exp)
  (caddr exp))

```

All that is left is to (define (make-exponentiation base exp)).

```

(define (make-exponentiation base exp)
  (cond ((=number? base 1) 1)
        ((=number? exp 1) base)
        ((=number? exp 0) 1)
        (else
         (list '** base exp))))

```

otakutyrant

Good simplification, but we can simplify it more:

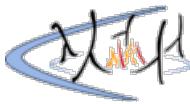
```

(define (make-exponentiation base exponent)
  (cond
    ((=number? base 1) 1)
    ((=number? exponent 0) 1)
    ((=number? exponent 1) base)
    ((and (=number? base) (=number? exponent)) (expt base exponent))
    (else (list '** base exponent)))
  )

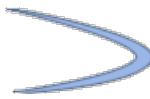
```

The above solution works. but in the end product, there is no need to check the exponent is number or not, procedure make-sum can do that. so the end product can be like this:

```
(define (deriv expr var)
  (cond ((number? expr) 0)
        ((variable? expr)
         (if (same-variable? expr var) 1 0))
        ((sum? expr)
         (make-sum (deriv (addend expr) var)
                   (deriv (augend expr) var))))
        ((product? expr)
         (make-sum
          (make-product (multiplier expr)
                        (deriv (multiplicand expr) var))
          (make-product (multiplicand expr)
                        (deriv (multiplier expr) var)))))
        ((exponentiation? expr)
         (make-product
          (make-product
           (exponent expr)
           (make-exponentiation (base expr)
                                 (make-sum (exponent expr) -1)))
          (deriv (base expr) var)))
        (else (error "unkown expression type -- DERIV" expr))))
```



# sicp-ex-2.57



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.56) | Index | Next exercise (2.58) >>

```
(define (make-sum-list l)
  (if (= (length l) 2)
      (list '+ (car l) (cadr l))
      (make-sum (car l) (make-sum-list (cdr l)))))

(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2)) (+ a1 a2))
        (else (make-sum-list (list a1 a2)))))

(define (make-product-list l)
  (if (= (length l) 2)
      (list '* (car l) (cadr l))
      (make-product (car l) (make-product-list (cdr l)))))

(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (number? m1) (number? m2)) (* m1 m2))
        (else (make-product-list (list m1 m2)))))

(define (augend s)
  (let ((a (cddr s)))
    (if (= (length a) 1)
        (car a)
        (make-sum-list a)))

(define (multiplicand p)
  (let ((m (cddr p)))
    (if (= (length m) 1)
        (car m)
        (make-product-list m)))

;; tests
(deriv '(* (* x y) (+ x 3)) 'x)
;; (+ (* x y) (* y (+ x 3)))

(deriv '(* x y (+ x 3)) 'x)
;; (+ (* x y) (* y (+ x 3)))
```

NTeGrotenhuis

Extend the differentiation program to handle sums and products of arbitrary numbers of (two or more) terms. Then the last example above could be expressed as

```
(deriv '(* x y (+ x 3)) 'x)
```

Try to do this by changing only the representation for sums and products, without changing the deriv procedure at all. For example, the addend of a sum would be the first term, and the augend would be the sum of the rest of the terms.

See [sicp-ex-2.56](#) for the differentiation program

All that must be done to solve this problem is to change augend and multiplicand so that they return the sum or product, respectively, of the remaining items in the list.

```
(define (augend s)
  (accumulate make-sum 0 (cddr s)))

(define (multiplicand p)
  (accumulate make-product 1 (cddr p)))
```

This works because (accumulate is really awesome.

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))
```

It recursively applies the make function to add up all the items in the list.

```
(accumulate make-sum 0 (cddr s))
```

Is analogous to

```
(accumulate + 0 (cddr s))
```

For symbolic data. We must use (cddr s) to get the rest of the list begining with the 3rd item.

meteorgan

As is mentioned above. All that must be done is to change "augend" and "multiplicand". The above solution is interesting, it use "accumulate" to change the representation of sum and product into the former version. for example: (augend (+ x x x x)) is (+ x (+ x x)), (multiplicand (\* x x x x)) is (\* x (\* x x)). But there is another solution. Here is my code:

```
;; if there is 4th item, make-sum 3rd item and 4th item
(define (augend expr)
  (if (null? (caddr expr))
      (caddr expr)
      (make-sum (caddr expr) (cadddr expr)))))

;; if there is 4th item, make-product 3rd item and 4th item
(define (multiplicand expr)
  (let ((first (caddr expr))
        (rest (cadddr expr)))
    (if (null? rest)
        first
        (make-product first (cadddr expr)))))
```

AMS

I agree with meteorgan's comment about the use of accumulate. As awesome and simple as it is, it changes the representation away from what we are using. I too came up with similar implementations of "augend" and "multiplicand" with the following changes. Instead of using "make-sum" and "make-product" I simply cons the operator to the beginning of the list.

```
(define (augend s)
  (if (null? (cdddr s))
      (caddr s)
      (cons '+ (cddr s)))))

(define (multiplicand p)
  (if (null? (cdddr p))
      (caddr p)
      (cons '* (cddr p)))))

;; using lets might make it easier to understand
(define (augend s)
  (let ((augend-element (cddr s)))
    (if (null? (cdr augend-element))
        (car augend-element)
        (cons '+ augend-element))))

(define (multiplicand p)
  (let ((multiplicand-element (cddr p)))
    (if (null? (cdr multiplicand-element))
        (car multiplicand-element)
        (cons '* multiplicand-element))))
```

electraRod

I agree with both AMS and meteorgan's solutions, as mine was quite similar, especially to meteorgan's. My solution is almost exactly the same as meteorgan's, with some slight difference that might make more intuitive sense (but not necessarily better). Instead of using caddr and cadddr to retrieve the 3rd and 4th items of the list, we can use some recursive thinking. Here is my code:

```
;; if there is no 3rd item, simply return the multiplicand.
;; Otherwise, the multiplicand is the product of the rest of the terms, ie, the product of
;; the multiplier of "cdr p" and the multiplicand of "cdr p"
(define (multiplicand p)
  (if (null? (cdddr p))
      (caddr p)
      (make-product (multiplier (cdr p)) (multiplicand (cdr p)))))

(define (augend s)
  (if (null? (cdddr s))
      (caddr s)
      (make-sum (addend (cdr s)) (augend (cdr s))))))
```

Note that these two procedures have very similar processes. In the spirit of SICP, we could define a HOP to capture this process.

AThird

Here's my attempt at this.

```
(define (augend s)
  (if (> (length s) 3)
      (make-sum (addend (cdr s)) (augend (cdr s)))
      (caddr s)))

(define (multiplicand p)
  (if (> (length p) 3)
      (make-product (multiplier (cdr p))
                    (multiplicand (cdr p)))
      (caddr p)))
```

Rptx

This code accomplishes the same results, but I added simplification to it.

```
(define (make-sum . l)
  (let ((lst (if (null? (cdr l)) (car l) l))) ;-> if it's a list inside a list, fix that.
    (let ((an (map (lambda (x) (if (and (pair? x) (sum? x))
                                         (make-sum (cdr x))
                                         x)) lst))) ;-> this will reduce inner sums.
      (let ((var-lst (filter (lambda (x) (not (number? x))) an))) ;-> make a list of
       variable
        (total (accumulate + 0 (filter number? an)))) ;-> sum all the numbers
      (cond ((null? var-lst) total) ;-> if there are no variables, than the sum is
           total
           ((= total 0) (if (null? (cdr var-lst)) ;-> if total is zero, then return
                           var-lst
                           (car var-lst) ;-> if it has only one element return it
                           (append (list '+) var-lst))) ;-> if it has more, then
           represent a
           ;list of the type (+ elements of var-lst)
           (else
            (append (list '+) ;-> else, just give a sum
                    var-lst
                    total))))))

(define (augend e)
  (make-sum (cddr e)))

; and make-product would be the same.

(define (make-product . l)
  (let ((lst (if (null? (cdr l)) (car l) l)))
    (let ((pn (map (lambda (x) (if (and (pair? x) (product? x))
                                      (make-product (cdr x))
                                      x)) lst)))
      (let ((var-lst (filter (lambda (x) (not (number? x))) pn)))
        (prod (accumulate * 1 (filter number? pn))))
      (cond ((null? var-lst) prod)
            ((= prod 1)
             (if (null? (cdr var-lst))
                 (car var-lst)
                 (append (list '* ) var-lst)))
            ((= prod 0) 0)
            (else
             (append (list '* )
                     var-lst
                     (list prod)))))))

(define (multiplicand p)
  (make-product (cddr p)))
```

fubupc

I have an version will "flatten" nested sum. e.g.

from:

```
(make-sum 'x 5 '(+ y 10) 'z)
```

to:

```
'(+ 15 x y z)
```

```

(define (make-sum . s)
  (define (sum-iter num-sum symbols seq)
    (if (null? seq)
        (cond ((null? symbols) num-sum)
              ((= num-sum 0))
              (if (= 1 (length symbols))
                  (car symbols)
                  (cons '+ symbols)))
              (else (cons '+ (cons num-sum symbols)))))
        (let ((next (car seq)))
          (rest (cdr seq)))
        (cond ((number? next) (sum-iter (+ num-sum next) symbols rest))
              ((and (pair? next) (sum? next)) (sum-iter num-sum symbols (append (cdr
next) rest)))
              (else (sum-iter num-sum (append symbols (list next)) rest))))))
  (sum-iter 0 '() s))

```

Genovia

this is my simple solution, and this is definitely right.

```

(define (augend s)
  (if (> (length s) 3)
      (cons '+ (caddr s))
      (caddr s)))

(define (multiplicand p)
  (if (> (length p) 3)
      (cons '*' (caddr p))
      (caddr p)))

```

Maanu

Both the solutions above with accumulators and cons symbols are concise and intuitive. In my primary solution, I in fact modified both of the constructor and selectors. But both parts of modifications are small and not difficult to understand.

```

(define (pure_pair? l)
  (and (pair? l)
       (not (sum? l))
       (not (product? l))
       (not (exponentiation? l)))))

(define (make-sum a1 a2)
  (cond ((pure_pair? a2)
         (make-sum a1 (make-sum (car a2) (cdr a2))))
        ((null? a2) a1) ;-> only these two new conditions are added to original make-
sum in order to work with list input
        ((and (number? a1) (= a1 0)) a2)
        ((and (number? a2) (= a2 0)) a1)
        ((and (number? a1) (number? a2)) (+ a1 a2))
        (else (list '+ a1 a2)))))

(define (make-product m1 m2)
  (cond ((pure_pair? m2)
         (make-product m1 (make-product (car m2) (cdr m2))))
        ((null? m2) m1) ;-> only these two new conditions are added to original make-
product in order to work with list input
        ((and (number? m1) (number? m2)) (* m1 m2))
        ((and (number? m1) (= m1 1)) m2)
        ((and (number? m1) (= m1 0)) 0)
        ((and (number? m2) (= m2 1)) m1)
        ((and (number? m2) (= m2 0)) 0)
        (else (list '* m1 m2)))))

(define (augend s) (make-sum (caddr s) (cdddr s))) ;-> augend would be the sum of the rest
of the terms
(define (multiplicand p) (make-product (caddr p) (cdddr p))) ;-> multiplicand would be the
product of the rest of the terms

```

In my second attempt, I realize that modifications only in selectors should be enough. I end up with new selectors as

```

(define (augend s)
  (define (process l)
    (if (null? (cdr l))
        (car l)
        (make-sum (car l) (process (cdr l)))))

  (process (cddr s)))

(define (multiplicand p)

```

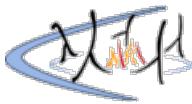
```
(define (process l)
  (if (null? (cdr l))
      (car l)
      (make-product (car l) (process (cdr l))))))
(process (cddr p))
```

When I was doing this, I gradually obtained a strong feeling that recursion should work in this case. I feel I should link the element of list with constructors (make-product / make-sum), together with recursive call of selector it-self (augend / multiplicand). However, I can not do it, because the selectors work with list starting with a sign (+/\*). Therefore, an additional function (process) is required to get rid of the sign during recursion. It turns out at the end this method is exactly the same as what accumulate dose. This deepen my understanding that with a higher level abstraction (accumulate), we change our mind of thinking about the subjects at hands and dealing with the problem in a more straightforward way. I still need more practices to get to that level of thinking. This kind of feeling of enlightenment, is just the value of doing SICP.

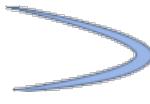
Sphinxsky

```
(define (augend s)
  (let ((other (caddr s)))
    (if (= (length other) 1)
        (car other)
        (cons '+ other)))))

(define (multiplicand p)
  (let ((other (cddr p)))
    (if (= (length other) 1)
        (car other)
        (cons '* other))))
```



# sicp-ex-2.58



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.57) | Index | Next exercise (2.59) >>

sgm

We're only going to bother with part (b) here, because the solution for that is also a solution for part (a).

The main problem is essentially to recognize whether a given expression is a sum or product. Now, keep in mind that, despite our moving to a representation that is more orthodox to traditional notation, we are still playing a pun: the parentheses which in one sense are used as mathematical groupings are at the same time sub-lists in list-structure. We can assume that these sub-lists will be valid expressions, so they will also be self-contained expressions. The upshot of this is that we need to concern ourselves with only the topmost “layer” of an expression: '(x \* y \* (x + z)) and '((x \* y) + (x \* y + z)) to give two examples, will look like '(x \* y \* █) and '(█ + █) as far as we're concerned.

Now to tell what sort of expression we have, we need to find out what operator will be the last one applied to the terms should we attempt to evaluate the expression. This has to be the operator with the lowest precedence among all the visible ones. So the predicates `sum?` and `product?` will search out the lowest-precedence operator and compare it to `+` and `*` respectively:

```
(define (sum? expr)
  (eq? '+ (smallest-op expr)))

(define (product? expr)
  (eq? '* (smallest-op expr)))
```

Where `smallest-op` searches an expression for the lowest-precedence operator, which can be done as an accumulation:

```
(define (smallest-op expr)
  (accumulate (lambda (a b)
    (if (operator? b)
        (minprecedence a b)
        a))
    'maxop
    expr))
```

There's a *lot* of wishful thinking going on here! Anyways, we need a predicate `operator?` which says if a symbol is a recognizable operator, `minprecedence` which is like `min` but over operator precedence instead of numbers, and a thing called `'maxop` which is basically a dummy value that is always considered “greater than” any other operator.

```
(define *precedence-table*
  '(
    (maxop . 10000)
    (minop . -10000)
    (+ . 0)
    (* . 1) )

(define (operator? x)
  (define (loop op-pair)
    (cond ((null? op-pair) #f)
          ((eq? x (caar op-pair)) #t)
          (else (loop (cdr op-pair)))))

  (loop *precedence-table*))

(define (minprecedence a b)
  (if (precedence<? a b)
      a
      b))

(define (precedence<? a b)
  (< (precedence a) (precedence b)))

(define (precedence op)
  (define (loop op-pair)
    (cond ((null? op-pair)
           (error "Operator not defined -- PRECEDENCE:" op))
          ((eq? op (caar op-pair))
           (cdar op-pair))
          (else
            (loop (cdr op-pair)))))

  (loop *precedence-table*))
```

So there is this thing we call the \*precedence-table\* which is a list of pairs mapping operator symbols to values denoting their absolute precedence; the higher the number, the higher the precedence. `operator?` is a search of this car's of the pairs, looking for a match. `min-precedence` orders two operators by the `operator<?` predicate, which tests if the precedence of the first operator is less than the second. `precedence` is a utility procedure to get the precedence value for an operator.<sup>1</sup>

So we can now recognize sums and products and the dispatching part of `deriv` now works. Let's now look at extracting their parts and making new ones. Given that `expr` is a list representing, say, a sum, then we can find the plus sign using `memq`. The augend of `expr` is the list of elements preceding the plus sign, and the addend the succeeding. Well, the augend is easy enough, it's the `cdr` of the result of `memq`:

```
(define (augend expr)
  (let ((a (cdr (memq '+ expr))))
    (if (singleton? a)
        (car a)
        a)))
```

N.B. The reason we test for a singleton (list of one element) and pull out the item is that otherwise `deriv` would be asked eventually to differentiate something like `(1)` or `('x)`, which it doesn't know how to do.

But to get the addend, we basically have to rewrite `memq`, but to accumulate the things prior to symbol.

```
(define (prefix sym list)
  (if (or (null? list) (eq? sym (car list)))
      '()
      (cons (car list) (prefix sym (cdr list)))))

(define (addend expr)
  (let ((a (prefix '+ expr)))
    (if (singleton? a)
        (car a)
        a)))
```

And now to make a sum, taking care of the standard numerical reductions:

```
(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2))
         (+ a1 a2))
        (else (list a1 '+ a2))))
```

And to finish things off, we'll define the procedures for products, which are basically similar to the above.

```
(define (multiplier expr)
  (let ((m (prefix '* expr)))
    (if (singleton? m)
        (car m)
        m)))

(define (multiplicand expr)
  (let ((m (cdr (memq '* expr))))
    (if (singleton? m)
        (car m)
        m)))

(define (make-product m1 m2)
  (cond ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((or (=number? m1 0) (=number? m2 0)) 0)
        ((and (number? m1) (number? m2))
         (* m1 m2))
        (else (list m1 '* m2))))
```

Well, let's take our new toy out for a test spin.

```
]=> (deriv '(x + 3 * (x + y + 2)) 'x)
;Value: 4

]=> (deriv '(x + 3) 'x)
;Value: 1

]=> (deriv '(x * y * (x + 3)) 'x)
;Value 88: ((x * y) + (y * (x + 3)))

;; Will extraneous parens throw our deriv for a loop?
]=> (deriv '(((x * y) * (x + 3)) 'x)
;Value 89: ((x * y) + (y * (x + 3)))

]=> (deriv '(x * (y * (x + 3))) 'x)
;Value 90: ((x * y) + (y * (x + 3)))
```

<sup>1</sup> I'm exposing the structure of the table by writing `caar` and such, but I'm sick enough of writing procedures not to bother with the proper abstraction layers (P.S. the wiki's Scheme highlighter doesn't understand `cdar`).

muff

I think sgm's (precedence op) procedure is wrong as it returns an error for non-operators, after searching the table, but instead it could return the max operator precedence:10000

AA

Part A is pretty straight forward, we will just change the representation of the date:

```
(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        (else (list a1 '+ a2)))))

(define (sum? x) (and (pair? x) (eq? (cadr x) '+)))
(define (addend s) (car s))
(define (augend s) (caddr s))

(define (make-product m1 m2)
  (cond ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((or (=number? m1 0) (=number? m2 0)) 0)
        (else (list m1 '* m2)))))

(define (product? x) (and (pair? x) (eq? (cadr x) '*)))
(define (multiplier x) (car x))
(define (multiplicand x) (caddr x))
```

For part B, since we can't use `cadr` to get the augend or the multiplicand because they might be more than one item, so we have to use `cddr`, but the problem with `cddr` is that it always returns a list, so we're going to use a cleaning procedure.

```
(define (cleaner sequence)
  (if (null? (cdr sequence))
      (car sequence)
      sequence))

(define (augend x)
  (cleaner (cddr x)))

(define (multiplicand x)
  (cleaner (cddr x)))
```

meteorgan

Here, we only think about sum expression and product expression. so if there is a '+' in the list, we think it's a sum expression, otherwise is a product expression. addend, multiplier is the part before '+', '\*' respectively in the list, augend, multiplicand is the part after '+', '\*' in the list. we only have to change predicates, selectors and constructors to solve the problem.

```
(define (operation expr)
  (if (memq '+ expr)
      '+
      '*))

(define (sum? expr)
  (eq? '+ (operation expr)))
(define (addend expr)
  (define (iter expr result)
    (if (eq? (car expr) '+)
        result
        (iter (cdr expr) (append result (list (car expr)))))))
  (let ((result (iter expr '())))
    (if (= (length result) 1)
        (car result)
        result)))
(define (augend expr)
  (let ((result (cdr (memq '+ expr))))
    (if (= (length result) 1)
        (car result)
        result)))

(define (product? expr)
  (eq? '*' (operation expr)))
(define (multiplier expr)
```

```

(define (iter expr result)
  (if (eq? (car expr) '*)
      result
      (iter (cdr expr) (append result (list (car expr))))))
(let ((result (iter expr '())))
  (if (= (length result) 1)
      (car result)
      result)))
(define (multiplicand expr)
  (let ((result (cdr (memq '* expr))))
    (if (= (length result) 1)
        (car result)
        result)))

```

brave one

one more option is to canonize (preprocess) given representation first, eg into fully parenthesized prefix notation. and \_then\_ do straightforward deriv computation. one advantage is that you probably can detect and transform number of representations. but haven't gotten to trying it out. otherwise @meteorgan's solution looks the best, simplest, even lacking generality.

fubupc

It seems @meteorgan 's answer has some topo error.

```

;; Parse error: Closing paren missing.
(define (addend expr)
  (define (iter expr result)
    (if (eq? (car expr) '+)

```

assume prefix expression ( '(+ x y) ) ?

Adam

I agree with "brave one", that the most appropriate option is to preprocess the representation first to provide precedence. This gives the advantage of separating concerns, keeping "deriv" simple and unchanged, and having a separate process for establishing precedence. This I think is in the spirit of how scheme works in general - the data requires a transformation that is easier to deal with. Anyway, here is my solution to parse the input before it is applied to "deriv":

```

(define (parse-precedence exp)

  (define (simplest-term? exp)
    (or (variable? exp) (number? exp)))

  (define (build-multiplier-precedence exp)
    (list (parse-precedence (multiplier exp))
          '*)
    (parse-precedence (multiplicand exp)))

  (define (iterate exp result)
    (cond ((null? exp) result)
          ((simplest-term? exp) exp)
          ((and (> (length exp) 2) (product? exp))
           (iterate (cdddr exp)
                   (cons (build-multiplier-precedence exp) result)))
          (else
            (iterate (cdr exp)
                    (cons (parse-precedence (car exp)) result)))))

  (iterate exp '()))

```

The above assumes we are only dealing with multiplication, and that the input is well formed.

The approach is to recursively parse each element, looking ahead for a product and applying the precedence accordingly. This is an O(n) process, with the advantage of applying simplicity to the problem, and separating the application of precedence from the problem of finding the derivative.

Pinlin(Calvin)-Xu

I did this a while ago and had the same idea of converting the notation from infix to prefix first in one pass, but all of the solutions were stack-based (possibly inspired by the shunting yard algorithm) instead of recursion-based that feels more natural for SICP. Anyways here is my solution in Racket that converts from infix to prefix notation recursively:

```

(define (precedence op)
  (cond
    [(eq? op '+) 0]
    [(eq? op '*) 1]
    [(eq? op '**) 2]))

```

```

(define (operator? x)
  (or (eq? x '+) (eq? x '* ) (eq? x '**)))

(define (min-operator expr)
  (accumulate (lambda (x y)
    (if (and (operator? x) (or (null? y) (< (precedence x) (precedence y)))) x y))
    nil
    expr))

(define (memq-before item x)
  (cond
    [ (null? x) nil]
    [ (eq? item (car x)) nil]
    [else (cons (car x) (memq-before item (cdr x)))]))

(define (before-op op expr)
  (unhusk (memq-before op expr)))

(define (after-op op expr)
  (if (pair? (memq op expr)) (cdr (memq op expr)) nil))

(define (rewrite expr)
  (let ([op (min-operator expr)])
    (define (helper remaining)
      (if (null? (after-op op remaining))
          (list (before-op op remaining))
          (cons (before-op op remaining) (helper (after-op op remaining))))))
    (cons op (helper expr)))))

(define (single? expr)
  (null? (cdr expr)))

(define (unhusk expr)
  (if (and (single? expr) (list? (car expr))) (unhusk (car expr)) expr))

(define (unwrap expr)
  (if (single? expr) (car expr) (cons (car expr) (flatmap identity (cdr expr)))))

(define (rewritten? expr)
  (or (single? expr) (operator? (car expr)))))

(define (complete? expr)
  (or (single? expr)
      (and (operator? (car expr))
           (pair? (cadr expr))
           (null? (filter (lambda (x) (not (single? x))) (cdr expr))))))

(define (to-prefix expr)
  (cond
    [ (not (rewritten? expr)) (to-prefix (rewrite expr))]
    [(complete? expr) (unwrap expr)]
    [else (cons (car expr) (map to-prefix (cdr expr))))]))
```

```

(to-prefix '(1 * x + y ** z + z * y * x))
(to-prefix '(1 * (2 + 3)))
(to-prefix '(2 ** (7 + (4 * 5))))
(to-prefix '(x + 3 * (x + y + 2)))

```

```

#+RESULTS:
: (+ (* 1 x) (** y z) (* z y x))
: (* 1 (+ 2 3))
: (** 2 (+ 7 (* 4 5)))
: (+ x (* 3 (+ x y 2)))

```

I really like how simple the recursive cases are in `to-prefix`; afterwards making `deriv` work is trivial

```

(define (deriv-infix exp var)
  (deriv (to-prefix exp) var))

```

Zelphir

I've got a solution for arbitrary structures of '+' and '\*' and '\*\*'. The idea is, to use precedence in reverse, so that operators with lower precedence will cause the terms to be split first and only then the operations of higher precedence. In order to calculate the derivation results of the lower precedence operations, the results from the operations with higher precedence are needed. Through recursion those of higher precedence will be the first ones to be calculated.

```

;; some helper procedure - there might be a library equivalent
(define (take-until a-list stop-elem)
  (define (iter result sublist)
    (cond
      [ (empty? sublist) result]

```

```

[ (eq? (car sublist) stop-elem) result]
[else (iter (append result (list (car sublist)))
             (cdr sublist)))]
(iter '() a-list))

```

## EXPLANATION

Important is the lowest precedence operation.

If the lowest precedence operation is found, it can be split up into its respective pairs of addend and augend, multiplier and multiplicand or base and exponent. The split will mean that the elements of the pairs are treated separately, which means that an operation of higher precedence will be treated separately, from operations of lower precedence, in a separate derivation call. Only when the derivates of subterms of higher precedence "bubble back up" in the recursive calls as return values, the subterms of lower precedence can be derived, because they rely on these results.

Example:

$(3 + 10 * x) ? \text{SUM } 3 \text{ AND } 10 * x$  | FIRST STEP

$(10 * x) ? \text{PRODUCT } 10 \text{ AND } x$  | SECOND STEP

The example shows that the precedence is used in reversed. This is reflected by the structure of the last-operation procedure.

```

(define (last-operation expression)
  (cond
    [ (memq '+ expression) '+]
    [ (memq '.* expression) '.*]
    [ (memq '** expression) '**]
    [else 'unknown]))

```

Then the predicates:

A term is a sum, if the operation of lowest precedence is a sum, because that means, that the last operation applied will be the sum.

```

(define (sum? expression)
  (and
    (list? expression)
    (eq? (last-operation expression) '+)))

(define (product? expression)
  (and
    (list? expression)
    (eq? (last-operation expression) '.*)))

(define (exponentiation? expression)
  (and
    (list? expression)
    (eq? (last-operation expression) '**)))

```

And the selectors for addend, augend, multiplier and multiplicand, base and exponent (note: I've not applied this concept to base and exponent, but it would work the same way):

```

(define (addend s)
  (let
    [(raw-addend (take-until s '+))]
    [if (= (length raw-addend) 1)
        (car raw-addend)
        raw-addend]))

(define (augend s)
  (let
    [(augend-part (cdr (memq '+ s)))]
    [if (= (length augend-part) 1)
        (car augend-part)
        augend-part]))

(define (multiplier product)
  (let
    [(raw-multiplier (take-until product '.*))]
    [if (= (length raw-multiplier) 1)
        (car raw-multiplier)
        raw-multiplier]))

(define (multiplicand p)
  (let
    [(multiplicand-part (cdr (memq '.* p)))]
    [if (= (length multiplicand-part) 1)
        (car multiplicand-part)
        multiplicand-part)])

```

```
(define (base power)
  (car power))

(define (exponent power)
  (caddr power))
```

And the make procedures, where I also changed some stuff:

```

(define (make-sum a1 a2)
  (cond
    [ (=number? a1 0) a2]
    [ (=number? a2 0) a1]
    [ (and
        (number? a1)
        (number? a2))
      (+ a1 a2)]
    [ (eq? a1 a2) (list 2 '* a1)]
    [ else
      (list a1 '+ a2)]))

(define (make-product m1 m2)
  (cond
    [ (or (=number? m1 0) (=number? m2 0)) 0]
    [ (=number? m1 1) m2]
    [ (=number? m2 1) m1]
    [ (and (number? m1) (number? m2))
      (* m1 m2)]
    [ else (list m1 '* m2)]))

(define (make-exponentiation base exponent)
  (cond
    [ (=number? exponent 1) base]
    [ (=number? exponent 0) 1]
    [ (=number? base 1) 1]
    [ (=number? base 0) 0]
    [ else (list base '** exponent)]))

```

vpraid

Here is my solution that utilizes modified shunting-yard algorithm, takes care of precedence and associativity, works in linear time, and is agnostic to implementation. I am only showing the algorithm itself and a few helpers, its application to derivate computation is trivial.

```

(and (eq? (associativity token) 'right)
      (< (precedence token)
           (precedence (car operators))))))
(iter (apply-op output (car operators)) (cdr operators) exp)
(iter output (cons token operators) (cdr exp)))
(else ; pushing new number or variable into output
      (iter (cons token output) operators (cdr exp))))))
(iter nil nil exp))

```

I am using `^` instead of `**` for exponentiation because it looks (to me, at least) better. Plus it has subtraction, but it is not hard to add it to the system.

Sphinxsky

My idea is that we can first convert the expression itself and convert the infix to a prefix, so that we don't have to rewrite the original code at all. Here is the code I wrote about the conversion expression.

```

;; Determine if it is an operator
(define (is-op? t)
  (or (eq? '+ t)
      (eq? '* t)
      (eq? '** t)))

;; Return operator priority
(define (order t)
  (cond ((eq? '+ t) 1)
        ((eq? '* t) 2)
        ((eq? '** t) 3)
        (else (error "Unknown symbol!"))))

;; return the corresponding operation according to the operator
(define (get-operator op)
  (cond ((eq? '+ op) make-sum)
        ((eq? '* op) make-product)
        ((eq? '** op) make-exponentiation)
        (else (error "Unknown symbol!"))))

(define (infix-to-prefix expr)

  (define (do-iter expr result-stack op-stack)
    (define (calculate expr op a1 a2 remain op-stack)
      (do-iter
        expr
        (cons ((get-operator op) a1 a2) remain)
        op-stack))

    (define (is-expression? x)
      (if (pair? x)
          (do-iter x '() '())
          x))

    (if (null? expr)
        (if (null? op-stack)
            (car result-stack)
            (calculate
              expr
              (car op-stack)
              (cadr result-stack)
              (car result-stack)
              (cddr result-stack)
              (cdr op-stack)))
        (let ((this (car expr))
              (other (cdr expr)))
          (if (is-op? this)
              (if (null? op-stack)
                  (do-iter other result-stack (cons this op-stack))
                  (let ((op-top (car op-stack))
                        (res-top (car result-stack)))
                    (if (>= (order op-top) (order this))
                        (calculate
                          expr
                          op-top
                          (cadr result-stack)
                          res-top
                          (cddr result-stack)
                          (cdr op-stack)))
                  (do-iter
                    other
                    result-stack
                    (cons this op-stack)))))))

```

```

(do-iter
  other
  (cons
    (is-expression? this)
    result-stack)
  op-stack)))))

(do-iter expr '() '()))

(define (prefix-to-infix expr)
  (define (do-it expr outer-order)
    (if (pair? expr)
        (let ((op (car expr)))
          (let ((this-order (order op)))
            (let ((a1 (cadr expr))
                  (a2 (caddr expr)))
              (if (> outer-order this-order) list (lambda (x) x))
              (append
                (do-it a1 this-order)
                (cons op (do-it a2 this-order)))))))
        (list expr)))
  (do-it expr (order '+)))

```

IAmParadox

An algorithm that creates brackets according to the precedence value of operators.  
Explanation.

Given an expression with more than 2 operations like

(x op1 y op2 z)

there will be at least one element such that Op1 y Op2 compare the precedence value of both operations

If Op1 > Op2 Resultant Expression will be

((x op1 y) op2 z)

If Op1 < Op2 Resultant Expression will be

(x op1 (y op2 z))

If Op1 = Op2 Resultant Expression will be

((x op1 y) op2 z)

If there's only one operation left in the expression we terminate. ((x op1 y) op2 z) which is the case here

As there's only op2 applied between (x op1 y) and z.

That's it.

Eg exp:

(x \* y + z \* ( m \* n + q ))

y is sandwiched between two \* and + . Since \* has higher precedence. Resultant expression will be

((x \* y) + z \* ( m \* n + q ))

Now the first element that is sandwiched between two operations is z. Since \* has higher precedence. Resultant expression

((x \* y) + ( z \* ( m \* n + q )))

But since it multiplicand is a pair we apply precedence sort to that

( m \* n + q )

Which will give

( (m \* n) + q )

So, the resultant expression will be

((x \* y) + (z \* ((m \* n) + q)))

Now we terminate because there's only one operation left ie + between ( $x * y$ ) and ( $z * ((m * n) + q)$ ).

This algorithm doesn't change any existing procedures. And to add a new op like  $\wedge$ . we just have to specify it's precedence value.

```
(define 1st-elem car)
(define op cadr)
(define 2nd-elem caddr)
(define 2nd-op cadddr)

(define (precedence-table exp)
  (cond
    ((eq? '+ exp) 0)
    ((eq? '* exp) 1)))

(define (precedence-check cmp exp)
  (cmp (precedence-table (op exp)) (precedence-table (2nd-op exp))))

(define (exp-f exp f)
  (list (f (1st-elem exp)) (op exp) (f (2nd-elem exp))))

(define (precedence-sort exp)
  (cond
    ((not (pair? exp)) exp)
    ((= (length exp) 3)
     (exp-f exp precedence-sort))
    ((precedence-check > exp)
     (precedence-sort (cons (exp-f exp precedence-sort) (cdddr exp))))
    ((precedence-check < exp)
     (append (list (precedence-sort (1st-elem exp)) (op exp))
             (list (precedence-sort (cddr exp))))))
    ((precedence-check = exp)
     (precedence-sort (cons (exp-f exp precedence-sort) (cdddr exp)))))))

(define (derivV2 exp var) (deriv (precedence-sort exp) var))
```

## Results

```
]=> (precedence-sort '(x * y + z * (m * n + q)))
; Result ((x * y) + (z * ((m * n) + q)))

]=> (derivV2 '(x * (y * (x + 3))) 'x)
; Result ((y * (x + 3)) + (x * y))
```

Sphinxsky

My idea is that we can first convert the expression itself and convert the infix to a prefix, so that we don't have to rewrite the original code at all. Here is the code I wrote about the conversion expression.

```
; Determine if it is an operator
(define (is-op? t)
  (or (eq? '+ t)
      (eq? '* t)
      (eq? '** t)))

;; Return operator priority
(define (order t)
  (cond ((eq? '+ t) 1)
        ((eq? '* t) 2)
        ((eq? '** t) 3)
        (else (error "Unknown symbol!"))))

;; return the corresponding operation according to the operator
(define (get-operator op)
  (cond ((eq? '+ op) make-sum)
        ((eq? '* op) make-product)
        ((eq? '** op) make-exponentiation)
        (else (error "Unknown symbol!"))))

(define (infix-to-prefix expr)

  (define (do-iter expr result-stack op-stack)
    (define (calculate expr op a1 a2 remain op-stack)
      (do-iter
```

```

expr
  (cons ((get-operator op) a1 a2) remain)
  op-stack)

(define (is-expression? x)
  (if (pair? x)
    (do-iter x '() '())
    x))

(if (null? expr)
  (if (null? op-stack)
    (car result-stack)
    (calculate
      expr
      (car op-stack)
      (cadr result-stack)
      (car result-stack)
      (cddr result-stack)
      (cdr op-stack)))
  (let ((this (car expr))
        (other (cdr expr)))
    (if (is-op? this)
      (if (null? op-stack)
        (do-iter other result-stack (cons this op-stack))
        (let ((op-top (car op-stack))
              (res-top (car result-stack)))
          (if (>= (order op-top) (order this))
            (calculate
              expr
              op-top
              (cadr result-stack)
              res-top
              (cddr result-stack)
              (cdr op-stack))
            (do-iter
              other
              result-stack
              (cons this op-stack))))))
      (do-iter
        other
        (cons
          (is-expression? this)
          result-stack)
        op-stack)))))

(do-iter expr '() '())

(define (prefix-to-infix expr)
  (define (do-it expr outer-order)
    (if (pair? expr)
      (let ((op (car expr)))
        (let ((this-order (order op)))
          (al (cadr expr))
          (a2 (caddr expr)))
        ((if (> outer-order this-order) list (lambda (x) x))
         (append
           (do-it al this-order)
           (cons op (do-it a2 this-order)))))))
      (list expr)))
  (do-it expr (order '+)))

```

x3v

Decided to write a pre-processing procedure instead, which recursively applies an "add-parens" procedure to a given expression. With this solution, adding support for exponentiation is trivial (see "process" procedure).

Nothing in the original implementations of deriv and its associated functions are changed. Solution partially inspired by regex. Suggestions on possible improvements are welcomed.

Edit: added "get-elem-idx-r" which gets index of last instance of element in sequence, just to accommodate edge cases in exponentiation expressions such as  $x^{**} 2^{**} 3^{**} 4$ . Shame because "get-elem-idx" is much cleaner.

```

;; Helper functions

(define (get-elem-idx s elem) ; gets idx of first instance of elem
  (if (eq? (car s) elem)
    0
    (+ 1 (get-elem-idx (cdr s) elem)))))

(define (get-elem-idx-r s elem) ; gets idx of last instance of elem
  (define (iter s idx counter)
    (cond ((eq? s '()) idx)

```

```

((eq? (car s) elem) (iter (cdr s) counter (+ 1 counter)))
  (else (iter (cdr s) idx (+ 1 counter)))))

(iter s 0 0))

(define (idx->elem s idx)      ;; gets element by index of a sequence
  (if (= idx 0)
    (car s)
    (idx->elem (cdr s) (- idx 1)))))

(define (range start end)      ;; same output as enumerate-interval
  (if (> start end) '()
    (cons start (range (+ start 1) end)))))

(define (add-parens s op)
  ;; returns an expression with brackets enclosing the nearest terms of a given operand
  (let ((idx (get-elem-idx-r s op)))
    (let ((left (- idx 1))
          (right (+ idx 1))
          (idxs (range 0 (- (length s) 1))))
      (let ((toloop (filter (lambda (x) (not (or (= x left) (= x right)))) idxs))
            (higher-order-expr (map (lambda (x) (idx->elem s x)) (range left right))))
        (map (lambda (x)
                  (if (= x idx) higher-order-expr (idx->elem s x)))
              toloop)))))

(define (process s)           ;; adds parens by order of operations
  (cond ((memq '** s) (process (add-parens s '**)))
        ((memq '*' s) (process (add-parens s '*)))
        (else (if (eq? '() (cdr s)) (car s) s)))))

(process '(y + x * y + x ** 2 ** 3 + 2 * 2)) ;; (y + (x * y) + (x ** (2 ** 3)) + (2 * 2))

(define deriv2 (lambda (exp var) (deriv (process exp) var)))
(deriv2 '(x * y + 1 + x ** 2 * 3 + 4) 'x) ;; (+ y (* (* 2 x) 3))

```

Andrey Portnoy

Part a. reusing our existing `deriv` implementation:

```

(define (infix->prefix exp)
  (if (not (pair? exp))
    exp
    (let ((a (car exp))
          (op (cadr exp))
          (b (caddr exp)))
      (list op (infix->prefix a) (infix->prefix b)))))

(define (prefix->infix exp)
  (if (not (pair? exp))
    exp
    (let ((op (car exp))
          (a (cadr exp))
          (b (caddr exp)))
      (list (prefix->infix a) op (prefix->infix b)))))

(define (deriv-infix exp var)
  (prefix->infix (deriv (infix->prefix exp) var)))

```

Part b. side-stepping the problem by converting free infix to strict infix to prefix and back:

```

;; helpers
(define (two l)
  (list (car l) (cadr l)))

(define (three l)
  (list (car l) (cadr l) (caddr l)))

;; parenthesize w.r.t. ops in the order they are passed to the procedure
(define (parenthesize exp op . ops)
  (define (iter head tail)
    (cond ((< (length tail) 4) (append head tail))
          ((eq? (cadr tail) op)
           (iter head
                 (cons (three tail) (cdddr tail))))
          (else
           (iter (append head (two tail))
                 (cddr tail)))))
  (let ((first
         (if (not (pair? exp))
           exp
           (map (lambda (e) (parenthesize e op))
```

```

        (iter '() exp)))))

(if (null? ops)
    first
    (apply parenthesize
           (append (list first (car ops))
                   (cdr ops)))))

(define (free-infix->prefix exp)
  (infix->prefix (parenthesize exp '* '+)))

(define (free-infix-deriv exp var)
  (prefix->infix (deriv (free-infix->prefix exp) var)))

```

chemPolonium

I first defined a infix->prefix function for part a, which can turn the infix notation into a prefix notation. Then a add-brackets function for part b, which can add missing brackets for the standard notation. And that works for me. (Note: I use racket, so some keywords may not be the same with scheme)

```

(define (infix->prefix exp)
  (cond [(number? exp) exp]
        [(variable? exp) exp]
        [else (list (cadr exp)
                    (infix->prefix (car exp))
                    (infix->prefix (caddr exp))))]))

(define (add-brackets exp)
  (cond ((number? exp) exp)
        ((variable? exp) exp)
        ((= (length exp) 1) (car exp))
        ((eq? (cadr exp) '*)
         (add-brackets (cons (add-brackets (car exp))
                             '*)
                           (add-brackets (caddr exp))))
         (caddr exp)))
        ((eq? (cadr exp) '+)
         (list (add-brackets (car exp))
               '+
               (add-brackets (cddr exp)))))
        [else (error "invalid function" exp)))))

(define (prefix->infix exp)
  (cond [(number? exp) exp]
        [(variable? exp) exp]
        [else (list (prefix->infix (cadr exp))
                    (car exp)
                    (prefix->infix (caddr exp))))]))

(add-brackets '(2 * 4 + 1))
(add-brackets '(2 + 4 * 1))
(add-brackets '(2 + 2 + 2))
(add-brackets '(x + 3 * (x + y + 2)))

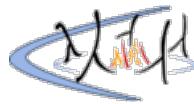
(infix->prefix (add-brackets '(x + 3 * (x + y + 2)))))

(define (deriv-infix exp var)
  (deriv (infix->prefix exp) var))
(define (deriv-std exp var)
  (deriv-infix (add-brackets exp) var))
(define (deriv-std->std exp var)
  (prefix->infix (deriv-std exp var)))

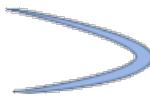
(deriv-std '(x + 3 * (x + y + 2)) 'x)
(deriv-std '(x + 3 * x * y + y + 2) 'x)

(deriv-std->std '(x + 3 * x * y + y + 2) 'x)
(deriv-std->std '(x + 3 * x * y * x + y + 2) 'x)

```



# sicp-ex-2.59



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.58) | Index | Next exercise (2.60) >>

Here's a tail-recursive implementation.

```
(define (union-set s1 s2)
  (if (null? s1)
      s2
      (let
        ((e (car s1)))
        (union-set
          (cdr s1)
          (if (element-of-set? e s2) s2 (cons e s2))))))

guile> (define element-of-set? member)
guile> (define odds '(1 3 5))
guile> (define evens '(0 2 4 6))

guile> (union-set '() '())
()

guile> (union-set '() evens)
(0 2 4 6)

guile> (union-set evens evens)
(0 2 4 6)

guile> (union-set evens odds)
(6 4 2 0 1 3 5)

guile> (union-set odds evens)
(5 3 1 0 2 4 6)

guile>
```

Lenatis

; ;Ex2.59

```
(define (union-set s1 s2)
  (cond ((and (null? s1) (not (null? s2)))
         s2)
        ((and (not (null? s1)) (null? s2))
         s1)
        ((element-of-set? (car s1) s2)
         (union-set (cdr s1) s2))
        (else (cons (car s1)
                     (union-set (cdr s1) s2)))))
```

Steve Zhang

; used filter to implement the intersection-set and union-set

```
(define (filter-intersection-set set1 set2)
  (filter (lambda (x) (element-of-set? x set2)) set1))

(define (filter-union-set set1 set2)
  (accumulate cons
              set2
              (filter (lambda (x) (not (element-of-set? x set2)))
                      set1)))
```

@bishboria

I thought of this nice solution using adjoin-set instead

```
(define (union-set set1 set2)
  (if (null? set1)
      set2
      (union-set (cdr set1) (adjoin-set (car set1) set2))))
```



Christian-Delahousse

Here's a solution using append and filter

```
(define (union s1 s2)
  (append s1 (filter (lambda (x)
                        (not (element-of-set? x s1)))
                      s2)))
```

Daniel-Amariei

Iterative and recursive processes.

```
; recursive process
(define (union-set A B)
  (cond ((null? B) A)
        ((element-of-set? (car B) A)
         (union-set A (cdr B)))
        (else (cons (car B)
                     (union-set A (cdr B))))))

;; iterative process
(define (union-set-iter A B)
  (define (iter A B U)
    (cond ((or (null? B) (null? A))
           U)
          ((element-of-set? (car B) A)
           (iter A (cdr B) U))
          (else (iter A (cdr B) (cons (car B) U)))))

  (iter A B A))
```

Lambdalef

Short solutions using adjoin-set

```
(define (union set1 set2)
  (accumulate adjoin-set set2 set1))
```

smatchcube

Note: be carefull with some of the solutions above, some of them are using the fact that no element appears more than once in set1 or set2 (respecting the representation of sets from the beginning of the chapter). For example bishboria's solution return (0 1 1) for (union-set '(0 0) '(1 1)). That's not really a problem in most cases but we must keep this in mind that we are assuming the input is correct, this is just a little trade-off for better performance.

Sphinxsky

I think this implementation is more complicated but the efficiency will be higher.

```
(define (union-set set1 set2)
  (define (do-iter set1 set2 result)
    (cond ((null? set1) result)
          ((null? set2) set1)
          (else
            (let ((this (car set1)))
              (do-iter
                (cdr set1)
                set2
                (if (element-of-set? this set2)
                    result
                    (cons this result)))))))
  (do-iter set1 set2 set2))
```

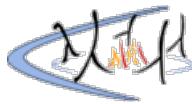
adsgray

I like the solutions that use accumulate.

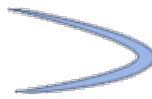
My solution is modeled on intersection-set and I think it is iterative. It accumulates into set2.  
It also assumes that the sets do not permit duplicate entries.

```
(define (union-set set1 set2)
  (cond ((null? set1) set2)
        ((null? set2) set1)
        ((element-of-set? (car set1) set2)
         (union-set (cdr set1) set2))
        (else (union-set (cdr set1) (cons (car set1) set2)))))
```

Last modified : 2021-05-14 18:56:04  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-2.60



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else (element-of-set? x (cdr set)))))

(define (adjoin-set x set)
  (cons x set))

(define (union-set set1 set2)
  (append set1 set2))

(define (remove-set-element x set)
  (define (remove-set-element-iter acc rest)
    (cond ((null? rest) acc)
          ((equal? x (car rest)) (append acc (cdr rest)))
          (else (remove-set-element-iter (adjoin-set (car rest) acc) (cdr rest)))))
  (remove-set-element-iter '() set))

(define (intersection-set set1 set2)
  (cond ((or (null? set1) (null? set2)) '())
        ((element-of-set? (car set1) set2)
         (cons (car set1)
               (intersection-set (cdr set1) (remove-set-element (car set1) set2))))
        (else (intersection-set (cdr set1) set2))))
```

<< Previous exercise (2.59) | Index | Next exercise (2.61) >>

sgm

With the need to check for pre-existence lifted, the procedure `adjoin-set` can be a straight `cons` and the time complexity drops to  $\Theta(1)$ .

```
(define (adjoin-set x set)
  (cons x set))
```

Also, `union-set` can now become a straight `append` and its time complexity drops to  $\Theta(n)$ .

```
(define (union-set set1 set2)
  (append set1 set2))
```

There isn't any way to improve `element-of-set?` or `intersection-set`.

bravely

@sgm, actually there is to improve `intersection-set`: getting all the dupes using union then filtering via strict intersection. strict version by itself only keeps dupes from the first set, not the second. didn't try to optimize though. `element-of-set` sure, stays as it is.

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else (element-of-set? x (cdr set)))))

(define (adjoin-set x set)
  (cons x set))

(define (intersection-set-strict set1 set2)
  (cond ((or (null? set1) (null? set2)) '())
        ((element-of-set? (car set1) set2)
         (cons (car set1)
               (intersection-set-strict (cdr set1) set2)))
        (else (intersection-set-strict (cdr set1) set2)))))

(define (intersection-set set1 set2)
  (let ((inter (intersection-set-strict set1 set2))
        (union (union-set set1 set2)))
    (filter (lambda (x) (element-of-set? x inter))
            union)))

(define (union-set set1 set2)
  (append set1 set2))
```

```
; from racket's rackunit
(check-equal? (intersection-set '(0 1 1 2 2 3) '(2 2 3 4))
               '(2 2 3 2 2 3))
```

Zelphir

@bravely I got another version for keeping the duplicates, although it might not be as elegant.

```
(define (element-of-duplicate-set? x set)
  (cond [(empty? set) false]
        [(equal? x (car set)) true]
        [else (element-of-duplicate-set? x (cdr set))]))

(define (adjoin-duplicate-set x set)
  (cons x set))

(define (intersection-duplicate-set set1 set2)
  (define (iter result s1 s2 already-swapped)
    (cond
      [(and (empty? s1) (empty? s2)) result]
      [(and (empty? s1) (not already-swapped))
       (iter result s2 result true)]
      [(and (empty? s1) already-swapped)
       result]
      [(element-of-duplicate-set? (car s1) s2)
       (iter (append result (list (car s1)))
             (cdr s1)
             s2
             already-swapped)]
      [else
       (iter result (cdr s1) s2 already-swapped))])
  (iter '() set1 set2 false))

(define (union-duplicate-set set1 set2)
  (append set1 set2))
```

sebastian

@bravely, if your improvement on intersection uses strict intersection, it has the same  $\Theta(n^2)$  complexity, so it is not an improvement at all. Even tho the definition might look cleaner, it is in fact actually worse than traversing the first set checking whether each element is also in the other set, because your improvement will need to append the sets in the union, so it will take  $n$  steps more.

Also, we haven't answered the last question in the exercise. You should used this non-strict repeated-allowing representation (which is just any data structure, really) every time you can. The *set* data structure is defined by this non-repeatedness and it should be used only where this property is necessary, because the set basic operations are expensive compared to other data structures.

master

### WARNING: This is wrong

-master

Here's how I implemented `element-of-set?` and `intersection-set`:

```
(define (remove x set)
  (filter (lambda (y) (not (equal? x y))) set))

(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else (element-of-set? x (remove (car set) (cdr set))))))

(define (intersection-set set1 set2)
  (cond ((or (null? set1) (null? set2)) '())
        ((element-of-set? (car set1) set2)
         (cons (car set1) (intersection-set (remove (car set1) (cdr set1)) set2)))
        (else (intersection-set (cdr set1) set2))))
```

I believe my `element-of-set?` is more efficient because it removes duplicates of negative matches while

it is searching. Imagine for example an ordered list with an initial member 1, a final member n which is what is being searched for and then n-2 other members each with an exorbitant amount of repetitions. If we don't filter those members which we know aren't in there then we will need to check them each time. This way we only need to check them once, turning something with quadratic time complexity into something with linearithmic time complexity (I think, time complexity isn't really my forte).

egbc

@master, unfortunately, your implementation of `element-of-set?` is quite inefficient. Since `remove` has an algorithmic complexity of  $\Theta(n)$ , and is potentially being called once for every element in the set when you call `element-of-set?`, you've created a procedure with an algorithmic complexity of  $\Theta(n^2)$ , whereas simply iterating over the full set without removing any elements would have a complexity of  $\Theta(n)$ . As an example, imagine the following procedure call:

```
(element-of-set? 11 (list 1 2 3 4 5 6 7 8 9 10))
```

`remove` will have been called 10 times, for no benefit.

master

You're right, there is no benefit; my thought process was mistaken. Although I do want to defend my implementation a little, yes for ordered sets with no repetition it is absolute rubbish. However, while it does have  $\Theta(n^2)$  worst case time complexity, if there are lots of repetitions then I think it is still  $\Theta(n)$  in the best case,  $\Theta(n \log n)$  average-case (correct me if I'm wrong). Obviously in this case that is indeed slower, but not as bad as  $\Theta(n^2)$ .

pilot227

I feel as though we're missing the point of the exercise. Please correct me if I'm wrong. But the point was to create operations that worked on sets that allow duplicates. So the intersection-set operation should return the duplicates from each set as well - returning the values which appear in both sets. Now admittedly there is some ambiguity here as to whether to include all common values (e.g if there are six 2s in set 1 and four 2s in set2 return four 2s) or all instances of common values (e.g if there are six 2s in set1 and four 2s in set2 return ten 2s) but this depends on what you're trying to do I suppose. Either way the remove-element procedures from above seem to be trying to impose the distinct element narrative from the previous exercise.

```
(define (intersection-set-dup set1 set2)
  (define (filter-set-dup predicate seq in out)
    (cond ((null? seq) (list in out))
          ((predicate (car seq))
           (filter-set-dup predicate (cdr seq) (cons (car seq) in) out))
          (else (filter-set-dup predicate (cdr seq) in (cons (car seq) out)))))

  (if (or (null? set1) (null? set2))
      '()
      (let ((filtererd_set1 (filter-set-dup (lambda (x) (equal? x (car set1))) set1
                                             '() '())))
        (filtererd_set2 (filter-set-dup (lambda (x) (equal? x (car set1))) set2
                                             '() '())))
        (append (append (car filtererd_set1) (car filtererd_set2))
                (intersection-set-dup (cadr filtererd_set1) (cadr
filtererd_set2)))))))
```

There are, no doubt, more elegant/efficient implementations. Say you were querying two databases and wanted all transactions above a certain value -- you'd want all duplicates (however they're defined - same value, same customer, same time, same item etc) to be present in the result.

partj

I think no nobody answered the last part of the question - when is the duplicate representation to be preferred? Here's my answer.

```
; Theta(n) but n could be large due to duplicates. Previously Theta(n) with no duplicates.
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else (element-of-set? x (cdr set)))))

; Theta(1). Previously Theta(n).
(define (adjoin-set x set) (cons x set))

; Theta(n). Previously Theta(n^2).
(define (union-set set1 set2) (append set1 set2))

; Theta(n^2) but n could be large due to duplicates. Previously Theta(n^2).
```

```
(define (intersection-set set1 set2)
  (cond ((or (null? set1) (null? set2)) '())
        ((element-of-set? (car set1) set2)
         (cons (car set1) (intersection-set (cdr set1) set2)))
        (else (intersection-set (cdr set1) set2))))
```

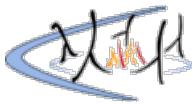
Observation: There is an efficiency improvement for operations that add to the set, while there's a drop in efficiency for the other operations (membership check and intersection). That drop is proportional to the no. of duplicates that are added to the set(s). e.g. For intersection of sets that are 2x their non-duplicate size, the performance cost is 4x.

For applications where the sets are to be constructed from known non-duplicate elements, this representation is more efficient than the non-duplicate one.

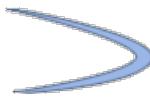
```
alice-packer
(define adjoin-set cons)
(define union-set append)
```

---

Last modified : 2022-05-05 15:29:05  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-2.61



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

```
(define (adjoin-set x set)
  (cond ((null? set) (list x))
        ((= x (car set)) set)
        ((< x (car set)) (cons x set))
        (else (cons (car set) (adjoin-set x (cdr set))))))
```

<< Previous exercise (2.60) | Index | Next exercise (2.62) >>

shyam

Exercise 2.61. Give an implementation of adjoin-set using the ordered representation. By analogy with element-of-set? show how to take advantage of the ordering to produce a procedure that requires on the average about half as many steps as with the unordered representation.

```
(define (adjoin-set x set)
  (cond ((or (null? set) (< x (car set))) (cons x set))
        ((= x (car set)) set)
        (else (cons (car set) (adjoin-set x (cdr set))))))
```

>>>

AMS

An iterative version of "adjoin-set".

```
(define (adjoin-set x set)
  (define (adjoin-set-iter early rest)
    (cond ((null? rest) (append early (list x)))
          ((= x (car rest)) (append early rest))
          ((< x (car rest)) (append early (cons x rest)))
          (else (adjoin-set-iter (append early (list (car rest)))
                                 (cdr rest)))))

  (adjoin-set-iter '() set))
```

Gera

This version runs in O(n<sup>2</sup>), as I believe append runs in O(n).>>>

m3tafunj

After doing exercise 2.61 it seemed like the exercise begged the question: how would you efficiently get all items that are not shared between sets, disjoin, or get all the items in set one that are not in set two, the difference? For anyone interested I wrote implementations for ordered lists that are O(n)

```
(define (disjoin set1 set2)
  ;numbers that are not in both sets
  (cond ((null? set1) set2)
        ((null? set2) set1)
        (else
         (let((x1 (car set1)) (x2 (car set2)))
           (cond((= x1 x2)
                 (disjoin (cdr set1) (cdr set2)))
                 ((< x1 x2)
                  (cons x1(disjoin (cdr set1) set2)))
                 (else(cons x2(disjoin set1 (cdr set2)))))))))

(disjoin '(1 2 3)'(1 3 4))
(disjoin '(1 2 3 6)'(1 3 4))
(disjoin '(1 2 3 6)'(1 3 4 5 7))

(define(difference set1 set2)
  ;only the numbers from the first set not in the second
  (cond((null? set1)'())
        ((null? set2)set1)
        (else
         (let((x1 (car set1)) (x2 (car set2))))
```

```

(cond( (= x1 x2)
        (difference (cdr set1) (cdr set2)))
      ((< x1 x2)
        (cons x1(difference(cdr set1)set2)))
      (else(difference set1 (cdr set2)))))))
(difference '(1 2 3)'(1 3 4))
(difference '(1 2 3 6)'(1 3 4))
(difference '(1 2 3 5 6)'(1 3 4 5 7))
(difference '(1 2 3 5 6 7)'(1 3 4 7))

```

horeszko

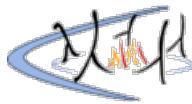
A version that handles ordered sets with duplicate elements in x or set.

```

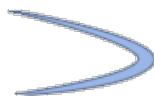
(define (adjoin-set x set)
  (cond ((null? set) (cons x set)) ; case: null set
        ((null? (cdr set)) (append set (list x))) ; case: end of list
        (else (let ((a (car set)) (b (car (cdr set))))) ; case: in list
               (cond ((> a x) (cons x set))
                     ((and (≤ a x) (≥ b x)) (append (list a x) (cdr set)))
                     ((< b x) (append (list a) (adjoin-set x (cdr set))))))))
; test

(adjoin-set 1 '())
(adjoin-set 1 '(2 3))
(adjoin-set 2 '(1 3))
(adjoin-set 1 '(1 2))
(adjoin-set 1 '(1 1))
(adjoin-set 4 '(1 2 3 5 6))
(adjoin-set 6 '(1 2 3 4 5 6))
(adjoin-set 7 '(1 2 3 4 5 6))

```



# sicp-ex-2.62



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.61) | Index | Next exercise (2.63) >>

shyam

Exercise 2.62. Give a  $\Theta(n)$  implementation of union-set for sets represented as ordered lists.

```
(define (union-set set1 set2)
  (cond ((null? set1) set2)
        ((null? set2) set1)
        ((= (car set1) (car set2))
         (cons (car set1) (union-set (cdr set1) (cdr set2))))
        ((< (car set1) (car set2))
         (cons (car set1) (union-set (cdr set1) set2)))
        (else
         (cons (car set2) (union-set set1 (cdr set2))))))
```

Or, If you want to make it ugly saving 5 car operations on a run,

```
(define (union-set set1 set2)
  (cond ((null? set1) set2)
        ((null? set2) set1)
        (else
         (let ((x1 (car set1))
               (x2 (car set2)))
           (cond ((= x1 x2) (cons x1 (union-set (cdr set1) (cdr set2))))
                 ((< x1 x2) (cons x1 (union-set (cdr set1) set2)))
                 (else (cons x2 (union-set set1 (cdr set2))))))))
```



Christian-Delahousse

Here's a slightly more elegant solution using adjoin-set from the previous exercise.

```
;From ex. 2.61
(define (adjoin-set x set)
  (cond ((null? set) (list x))
        ((= x (car set)) set)
        ((> x (car set))
         (cons (car set) (adjoin-set x (cdr set))))
        ((< x (car set))
         (cons x set))
        (else (error "WTF!?" ))))

;Answer
(define (union-set s1 s2)
  (cond ((null? s1) s2)
        ((element-of-set? (car s1) s2)
         (union-set (cdr s1) s2))
        (else (union-set (cdr s1) (adjoin-set (car s1) s2)))))
```

Gera

This solution is not  $O(n)$  though. adjoin-set and union-set are  $O(n)$ . If you call either one for each  $n$ , the solution becomes  $O(n^2)$ .

Here's my solution. Is basically the same as the first one, but more elegant.

```
(define (union-set-2 set1 set2)
  (cond ((null? set1) set2)
        ((null? set2) set1)
        (else
         (let ((x1 (car set1))
               (x2 (car set2)))
           (cons (min x1 x2)
                 (union-set-2 (if (> x1 x2)
```

```

        set1
        (cdr set1))
(if (> x2 x1)
set2
(cdr set2)))))))

```

brave one

racket matching, not as pretty as haskell, but cool anyway

```

(define/match (union-set set1 set2)
[('() set2) set2]
[(set1 '()) set1]
[((list x1 rest1 ...) (list x2 rest2 ...))
(cond [(< x1 x2) (cons x1 (union-set rest1 set2))]
[(< x2 x1) (cons x2 (union-set set1 rest2))]
[else (cons x1 (union-set rest1 rest2))])])

```

Zelphir

I've created a mess ... but it seems to work. It is tail recursive and builds a new resulting list from the two lists / sets. Not sure if tail recursive makes sense here though, since we are building a recursive structure. Anyway, here it is:

```

(define (union-ordered-set set1 set2)
(define (iter result subset1 subset2)
(cond
[ (and (empty? subset1) (empty? subset2))
result]

[ (empty? subset1)
(cond
[ (= (car subset2) (car result)) (iter result subset1 (cdr subset2))]
[else (iter (cons (car subset2) result) subset1 (cdr subset2)))] ; is always
the > case, because smaller elements get added first!

[ (empty? subset2)
(cond
[ (= (car subset1) (car result)) (iter result (car subset1) subset2)]
[else (iter (cons (car subset1) result) (cdr subset1) subset2)]) ; is always
the > case
[else ; set1 and set2 are not empty
(cond
[ (empty? result) ; initial case
(cond
[ (= (car subset1) (car subset2))
(iter (cons (car subset1) result) (cdr subset1) (cdr subset2)))

[ (> (car subset1) (car subset2))
(iter (cons (car subset2) result) subset1 (cdr subset2))]

[else ; set1_1 < set2_1
(iter (cons (car subset1) result) (cdr subset1) subset2)])]

[(> (car subset1) (car subset2))
(iter (cons (car subset2) result) subset1 (cdr subset2))]

[(< (car subset1) (car subset2))
(iter (cons (car subset1) result) (cdr subset1) subset2)]

[else ; both first elements are equal
(cond
[ (> (car subset1) (car result))
(iter (cons (car subset1) result) (cdr subset1) (cdr subset2))]
[else
(iter result (cdr subset1) (cdr subset2))))]]]
(reverse (iter nil set1 set2)))

```

I know it's ugly. I probably should use a let and avoid a lot of the cars. Here are some unit tests:

```

(test-case
"union-ordered-set test case"
(check-equal?
(union-ordered-set '(1 2 4) '(3))
'(1 2 3 4)
"union-ordered-set does not work correctly")
(check-equal?
(union-ordered-set '(1 2) '(1 3))
'(1 2 3)
"union-ordered-set does not work correctly")

```

```
(check-equal?
  (union-ordered-set '(2 4 9) '(1 4 9))
  '(1 2 4 9)
  "union-ordered-set does not work correctly"))
```

jammy

```
(define (union-set-2 set1 set2)
  (cond ((null? set1) set2) ((null? set2) set1)
        (else (let ((a (car set1)) (s1- (cdr set1))
                    (b (car set2)) (s2- (cdr set2)))
                (cond ((< a b) (cons a (union-set-2 s1- (cons b s2-))))
                      ((= a b) (cons a (union-set-2 s1- s2-)))
                      ((< b a) (union-set-2 (cons b s1-) (cons a s2-))))))))
```

the let here is roughly trying to simulate pattern matching. the solution is similar to others, the most unique thing is the line ((< b a) (union-set-2 (cons b s1-) (cons a s2-))) which swaps the first elements of each list so that the smallest element is the car of the first argument, this then activates the (< a b) case on the next call.

## category-learning-scheme

Last modified : 2023-11-02 01:59:34  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-2.63

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.62) | Index | Next exercise (2.64) >>

skangas

a. The trees shown in figure 2-16 can be written as:

```
(define fig2-16-1 '(7 (3 (1 () ()) (5 () ())) (9 () (11 () ())))  
(define fig2-16-2 '(3 (1 () ()) (7 (5 () ())) (9 () (11 () ())))))  
(define fig2-16-3 '(5 (3 (1 () ()) ()) (9 (7 () ()) (11 () ())))))
```

Using either function, the trees all evaluate to the same list:

```
(1 3 5 7 9 11)
```

b. The first function is of  $O(n \log n)$  complexity. The second function is of  $O(n)$  complexity.

meteorgan

a. Both procedures produce the same result because they both are in-order traversals.

b. Let  $T(n)$  be the time taken by the procedure for a balanced tree with  $n$  nodes.

For tree->list-1:

$T(n) = 2*T(n/2) + O(n/2)$  (as the procedure `append` takes linear time)  
Solving above equation, we get  $T(n) = O(n * \log n)$

For tree->list-2:

$T(n) = 2*T(n/2) + O(1)$

Solving the above equation, we get  $T(n) = O(n)$

db

\*by Master Theorem.

wl

I cannot see why the time complexity of the two implementations are different. If we ignore the specific implementations and only consider it from an abstract algorithmic perspective, both are  $O(N)$  tree traversal.

Of course one can then argue that `append` is  $O(N)$  while `cons` is  $O(1)$ , which I suppose is correct. (But I am not one hundred percent sure about that. I would appreciate if anybody can give reference to the built-in implementation of `append`, preferably to the Racket interpreter, which I am using). So `tree->list-1` now becomes  $O(N \log N)$ .

But `tree->list-2` might also be  $O(N \log N)$ . Consider the third formal argument of `copy-to-list`, i.e. `result-list`. Again I know little about the call stack or the details of Racket/Scheme interpreter(s), but does this require  $O(N)$  time and space to allocate and copy the data? If it does, the merge step of Divide and Conquer would be  $O(n)$  as well. As a result, the time complexity would be  $O(N \log N)$ .

master

I'm not 100% sure about my answer but I spent some time tracing out the procedures by hand and it is pretty strikingly obvious how much pointless work is being done by `tree->list-2`. The difference between the two procedures basically boils down to the following difference:

`tree->list-1` fully expanded

```
(cons 1 (cons 3 (cons 5 (cons 7 (cons 9 (cons 11 '()) ()))))
```

`tree->list-2` fully expanded (I think it may depend on the tree structure. this is tree A i.e. (7 (3 (1 () ()) (5 () ())) (9 () (11 () ())))

```
(append  
  (append (append '() (cons 1 '())) (cons 3 (append '() (cons 5 '()))))  
  (cons 7 (append '() (cons 9 (append '() (cons 11 '()))))))
```

I got a little lazy with the substitution because I had already done all the trees for `tree->list-1`,

but I think this is more or less correct. The point is, it does pointless things like appending to the empty list and worst of all the very last thing it does is append the right branch of the whole tree to its left branch. This is where the procedures differ most dramatically; They take about as much time and recursion depth to set up the final operation I think, but `tree->list-1` can build the list in linear time, whereas `tree->list-2` has to go through the left branch twice, once to append and cons it up, and then again to locate the end of the list. Anyway, the procedures do the exact same thing except that `tree->list-2` takes at least twice as long to do it.

# sicp-ex-2.64

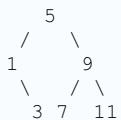
[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.63) | Index | Next exercise (2.65) >>

- a. PARTIAL-TREE splits the list ELTS into three parts: the median item THIS-ENTRY, the list of items less than the median, and the list of items greater than the median. It creates a binary tree whose root node is THIS-ENTRY, whose left subtree is the PARTIAL-TREE of the smaller elements, and whose right subtree is the PARTIAL-TREE of the larger elements.



- b. At each step, PARTIAL-TREE splits a list of length  $n$  into two lists of approximate length  $n/2$ . The work done to split the list is (QUOTIENT (- N 1) 2) and (- N (+ LEFT-SIZE 1)), both of which take constant time. The work to combine the results is (MAKE-TREE THIS-ENTRY LEFT-TREE RIGHT-TREE), and is also constant. Therefore, the time to make the partial tree of a list of  $n$  elements is:

$$T(n) = 2T(n/2) + \Theta(1)$$

By the Master Theorem, we have  $a = 2$ ,  $b = 2$ , and  $f(n) = \Theta(1)$ . Therefore,  $T(n) = \Theta(n)$ .

The time taken by LIST->TREE for a list of length  $n$  will be the time taken by PARTIAL-TREE plus the time taken by LENGTH for that list. Both procedures have order of growth  $\Theta(n)$ , so the order of growth of LIST->TREE is  $\Theta(n)$ .

atomik

I got a different answer by setting breakpoints in the dracket debugger and stepping through the program. The number of stack frames hovered around  $\log_2(n)$ , and the `partial-tree` function was recursively called roughly  $2^n$  number of times. That lead me to believe that the orders of growth are:

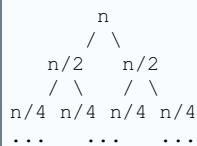
Space:  $O(\log_2(n))$

Time:  $O(2^n)$

I realize my answer is very different from the one given above. Did I do something wrong?

anonymous

To atomik. It is certain that the procedure `parital-tree` is called  $O(2^d)$  times with respect to the **depth d** of the calling tree. However, in terms of the **size of the input n**, the calling tree looks like:



There are  $O(2n-1)$  nodes, and the cost is  $O(1)$  at each node. Therefore, the total cost is  $\Theta(n)$ .

thanhnghuyen2187

I found it easier explaining the whole process as steps:

- Split the elements into left half, middle element, and right half
- Build the left tree with the left half
- Build the right tree with the right half
- Return a built tree with the middle element, the left tree, and the right tree

Here is a result table for easier referencing (please copy the text to a wider screen, or to a Markdown previewer):

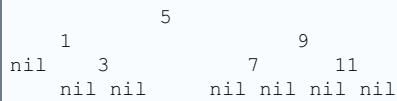
n   elts	left-size	left-result	left-tree	non-left-elts	
right-size	right-result	right-tree	left-tree	non-left-elts	-----
-	-----	-----	-----	-----	-----

1   (3 5 7 9 11)   0	()   ((3 5 7 9 11))   0	()   (3 5 7 9 11)   0
()   (5 7 9 11)   0	()   ((5 7 9 11))   0	()   (5 7 9 11)   0
2   (1 3 5 7 9 11)   0	(3)   ((1 3 5 7 9 11))   1	(1 3 5 7 9 11)   1
((3) () ())   0	(3)   ()   (3 5 7 9 11)   0	(1 3 5 7 9 11)   0
1   (7 9 11)   0	()   ((7 9 11))   0	(7 9 11)   0
()   (9 11)   0	()   ()   (9 11)   0	()   (9 11)   0
1   (11)   0	()   ((11))   0	()   (11)   0
()   ()   1	((7) () ())   0	(7) () ()   1
((11) () ())   0	(11)   ()   (9 11)   0	(9 11)   0
6   (1 3 5 7 9 11)   2	((1) () (3) () ())   0	(1) () (3) () ()   3
((9) (7) () ())   0	(9)   (7)   (11)   0	(5 7 9 11)   0

jirf

a.

The *partial-tree* function relies on the ease of calculating the trivial case of a tree containing zero elements, and the ability to calculate the size of left/right sub-trees with mere arithmetic. The function recurses first down the left sub-trees by (*partial-tree* ents left-size). When n=1, left/right-size=0 so left/right-result= ('() . ents), this-entry=(car ents), remaining-ents=(cdr ents). All are correct. This correct left-result means the calling function receives the correct left-tree, remaining-ents, and can correctly calculate right tree with (*partial-tree* remaining-ents right-size). As with the next calling function, ad infinitum.



b.

I believe that *partial-tree* is O(n) but math is not my strong suite and I found this quite tricky

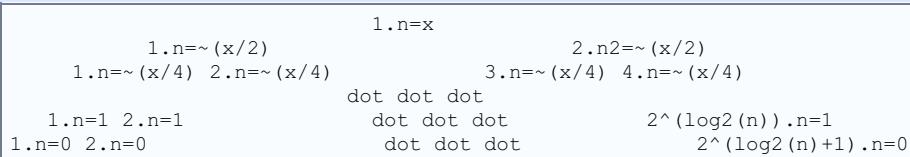
Assuming that elts is at least n-elements long the number of steps to calculate the result of any call to *partial-tree* is dependent on n.

For any n > 0 calculating (*partial-tree* elts n) requires calculating

```
(partial-tree elts (quotient (- n 1) 2))
(partial-tree (cd..dr elts) (- n (+ left-size 1)))
```

The new n parameters are ~n/2.

Here is a tree showing the structure of the recursive calls.

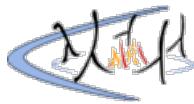


The height of this pyramid is ~log2(n)+1 because n (basically) halves 2 twice at each stage until it hits n=1, then it becomes 0 twice. To calculate the total number of calls to *partial-tree* we notice we are adding the widths of the ~log2(n)+1 levels of the pyramid.

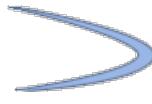
This boils down to (sum from=0 to=-log2(n)+1 k=2^x). If I remember correctly the answer to that summation of 2^x from 0 to x is 2^(x+1) - 1. Therefore the number of steps is

- O(~2^(log2(n)+2)-1) ->
- O(~2^(log2(n))\*(2^2)) ->
- O(~n^4) ->
- O(n)

[<< Previous exercise \(2.63\)](#) | [Index](#) | [Next exercise \(2.65\) >>](#)



# sicp-ex-2.65



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.64) | Index | Next exercise (2.66) >>

hp

I don't see how to do this for arbitrary sets without recurring to allowing multiple copies of an element (using the O(n) from 2.60 for instance) i.e. using multisets.

```
; ; we use union-set260 which is union for multisets and O(n).  
; ; As both conversions tree->list2 (2.63) and list->tree are O(n) we are good  
; ; NOTE THAT resulting tree is a multiset  
(define (union-set265 s t)  
  (list->tree (union-set260 (tree->list-2 s)  
                           (tree->list-2 t))))
```

tig

leafac's algorithm for union-set does not work properly. If, for instance, the entry of one tree is bigger than the other, it won't compare the right branch of the tree with the bigger entry with the tree with the smaller entry. Try this:

```
(union-set (list->tree '(1 2 3 5 7 9 46))  
          (list->tree '(5 6 10 11 20 23 46)))  
  
; ; => (11 (6 (5 (2 (1 () ()) (3 () ()) (9 (7 () ()) (46 () ())))) (10 () ())) (23 (20 () (46 () ())))
```

46 shows up twice.

leafac

If you want to avoid the conversions back and forth from representations, you can choose to don't use the results of exercises 2.63 and 2.64:

```
(define (union-set a b)  
  (cond ((null? a) b)  
        ((null? b) a)  
        (else  
         (let ((a-entry (entry a))  
               (a-left-branch (left-branch a))  
               (a-right-branch (right-branch a))  
               (b-entry (entry b))  
               (b-left-branch (left-branch b))  
               (b-right-branch (right-branch b)))  
          (cond ((= a-entry b-entry)  
                  (make-tree a-entry  
                             (union-set a-left-branch b-left-branch)  
                             (union-set a-right-branch b-right-branch)))  
                ((< a-entry b-entry)  
                 (make-tree b-entry  
                            (union-set a b-left-branch)  
                            b-right-branch))  
                ((> a-entry b-entry)  
                 (make-tree a-entry  
                            (union-set a-left-branch b)  
                            a-right-branch)))))))  
  
(union-set (list->tree '(1 3 5))  
          (list->tree '(2 3 4)))  
; ; => (3 (2 (1 () ()) ()) (5 (4 () ()) ()))  
  
; warning this algo has wrong output  
; ; check input as bst1 bst2 as:  
; (define bst1 (list->tree '(1 2 3 4 5 6 7)))  
; (define bst2 (list->tree '(6 7 8 9)))  
(define (intersection-set a b)  
  (cond ((null? a) ())  
        ((null? b) ())  
        (else  
         (let ((a-entry (entry a))  
               (a-left-branch (left-branch a))
```

```

(a-right-branch (right-branch a))
(b-entry (entry b))
(b-left-branch (left-branch b))
(b-right-branch (right-branch b)))
(cond ((= a-entry b-entry)
       (make-tree a-entry
                  (intersection-set a-left-branch b-left-branch)
                  (intersection-set a-right-branch b-right-branch)))
      ((< a-entry b-entry)
       (union-set
         (intersection-set a-right-branch
                           (make-tree b-entry () b-right-branch))
         (intersection-set (make-tree a-entry a-left-branch ())
                           b-left-branch)))
      ((> a-entry b-entry)
       (union-set
         (intersection-set (make-tree a-entry () a-right-branch)
                           b-right-branch)
         (intersection-set a-left-branch
                           (make-tree b-entry b-left-branch ())))))
(intersection-set (list->tree '(3 5 10))
                  (list->tree '(1 2 3 4 5 7)))
;; => (5 (3 () ()) ())

```

ljp

### anohter solution

```

(define (addjoin x set)
  (cond ((null? set) (make-tree x '() '()))
        ((= x (entry set)) set)
        ((< x (entry set)) (make-tree (entry set) (addjoin x (left-branch set)) (right-
branch set)))
        ((> x (entry set)) (make-tree (entry set) (left-branch set) (addjoin x (right-
branch set))))))
(define (element-of? x set)
  (cond ((null? set) false)
        ((= x (entry set)) true)
        ((< x (entry set)) (element-of? x (left-branch set)))
        ((> x (entry set)) (element-of? x (right-branch set)))))
(define (union-set set1 set2)
  (cond ((null? set1) set2)
        ((null? set2) set1)
        (else
          (let ((result-entry (addjoin (entry set1) set2)))
            (let ((left-result (union-set (left-branch set1) result-entry)))
              (union-set (right-branch set1) left-result))))))
(define (intersection-set set1 set2)
  (cond ((null? set1) '())
        ((null? set2) '())
        ((element-of? (entry set1) set2)
         (make-tree (entry set1)
                    (intersection-set (left-branch set1) set2)
                    (intersection-set (right-branch set1) set2)))
        (else
          (union-set (intersection-set (left-branch set1) set2)
                     (intersection-set (right-branch set1) set2)))))

;next is another list->tree
(define (sub-lst lst a b)
  (define (iter lst index a b result)
    (cond ((or (null? lst) (> index b)) result)
          ((< index a) (iter (cdr lst) (+ index 1) a b result))
          (else
            (iter (cdr lst) (+ index 1) a b (append result (list (car lst)))))))
  (iter lst 0 a b '()))
(define (list->tree2 lst)
  (let* ((n (length lst))
         (part (quotient (- n 1) 2)))
    (cond ((= n 0) '())
          ((= n 1) (make-tree (car lst) '() '()))
          (else
            (let ((left-part (sub-lst lst 0 (- part 1)))
                  (right-part (sub-lst lst part n)))
              (let ((entry (car right-part)))
                (make-tree entry
                           (list->tree2 left-part)
                           (list->tree2 (cdr right-part)))))))))


```

emj

set and intersection-set for sets implemented as (balanced) binary trees. I don't see that in any of these solutions. I am renaming tree->list-2 (from the course text) to tree->list.

```

(copy-to-list tree '()))

(define (max x1 x2)
  (if (> x1 x2) x1 x2))

;; ****Testing*****
(depth (union-set-bal (list->tree '(1 2 3 5 7 9 46)) (list->tree '(5 6 10 11 20 23 46))))
;;4
(depth (intersection-set-bal
  (list->tree '(1 3 5 7 10 14 15 20 31 50 51 53 55 57 59 61 63 65))
  (list->tree '(1 3 5 8 10 24 25 30 41 50 52 54 56 58 60 62 64 66))))
;;3

;; *****Used for testing*****
(define (depth tree)
  (if (null? tree)
    0
    (+ 1 (max (depth (car (cdr tree)))
               (depth (car (cdr (cdr tree))))))))

```

cgb

Can't we just use intersection-set and union-set from ex. 2.60 and 2.62? In this case we would just first convert our trees using tree->list-2 and, after the procedure, convert the lists again using list->tree, as the exercise statement sort of implies when it asks to "use the results of exercises 2.63 [tree->list] and 2.64 [list->tree]". I just adapted the code from these exercises, but I'm not sure if this implementation has an order of growth O(n)

```

(define (union-set set1 set2)
  (define (aux set1 set2)
    (cond ((null? set2) set1)
          ((null? set1) set2)
          ((= (car set1) (car set2))
           (cons (car set1) (aux (cdr set1) (cdr set2)))))
          ((< (car set1) (car set2)) (cons (car set1) (aux (cdr set1) set2)))
          (else (cons (car set2) (aux set1 (cdr set2))))))
  (list->tree (aux (tree->list-2 set1) (tree->list-2 set2)))))

(define (intersection-set set1 set2)
  (define (aux set1 set2)
    (cond ((or (null? set1) (null? set2)) '())
          ((= (car set1) (car set2))
           (cons (car set1) (aux (cdr set1) (cdr set2)))))
          ((< (car set1) (car set2))
           (aux (cdr set1) set2))
          ((> (car set1) (car set2))
           (aux set1 (cdr set2)))))
  (list->tree (aux (tree->list-2 set1) (tree->list-2 set2)))))


```

dave

This is how I did it. Using **tree->list2** from ex2.63 and **list->tree** from ex2.64. **union-set** is O(n) because it does one **cons** call for every number in the resulting set which, worst case, is the length of set1 plus the length of set2. It also calls **list->tree** once and **tree->list** twice. So we have O(n) + O(n) + 2\*O(n).

Similar reasoning can be applied to **intersection-set**.

```

(define (union-set set1 set2)
  (define (union-list set1 set2)
    (cond ((null? set1) set2)
          ((null? set2) set1)
          (else (let ((x1 (car set1)) (x2 (car set2)))
                  (cond ((equal? x1 x2)
                         (cons x1 (union-list (cdr set1) (cdr set2))))
                         ((< x1 x2)
                          (cons x1 (union-list (cdr set1) set2)))
                         ((< x2 x1)
                          (cons x2 (union-list set1 (cdr set2))))))))
    (list->tree (union-list (tree->list set1) (tree->list set2)))))

(define (intersection-set set1 set2)
  (define (intersection-list set1 set2)
    (cond ((null? set1) '())
          ((null? set2) '())
          (else (let ((x1 (car set1)) (x2 (car set2)))
                  (cond ((equal? x1 x2)
                         (cons x1 (intersection-list (cdr set1) (cdr set2))))
                         ((< x1 x2)
                          (intersection-list (cdr set1) set2)))))))
  (list->tree (intersection-list (tree->list set1) (tree->list set2)))))


```

```

        ((< x2 x1)
         (intersection-list set1 (cdr set2))))))
(list->tree (intersection-list (tree->list set1) (tree->list set2))))

```

jirf

## Solution

My solution relies on the fact that we can traverse trees in order in O(n) time. By transforming the trees into a *list-like* form, (*head . tail*), we can reduce the problem to a linear recursive process of comparing each set's "head".

### Note 1

Sorry for the length of this...

### Note 2

My solution is a little dense for a couple of reasons

1. My code operates on any number of sets, not just 2 (IDK why I did that)
2. I wanted to avoid creating a bunch of redundant lists so I created a function that would traverse a tree in something of the fashion of a Python iterator. These algorithms would work on list representations of tree's just as well.

## Helper Functions

```

(define (fold-left op initial sequence)
  (define (iter seq result)
    (if (null? seq)
        result
        (iter (cdr seq)
              (op (car seq)
                  result))))
  (iter sequence initial))

(define (filter predicate? sequence)
  (cond
    ((null? sequence) nil)
    (else
      (if (predicate? (car sequence))
          (cons (car sequence)
                (filter predicate? (cdr sequence)))
          (filter predicate? (cdr sequence)))))))

(define (any? predicate? elements)
  ;;; cause (apply or (list blah blah)) does not work
  (fold-left
    (lambda (left right)
      (or (predicate? left) right))
    #f
    elements))

(define (all? predicate? elements)
  ;;; cause (apply and (list blah blah)) does not work
  (fold-left
    (lambda (left right)
      (and (predicate? left) right))
    #t
    elements))

(define (remove-nulls elements)
  ;;; guess what this does :
  (filter (lambda (x) (not (null? x))) elements))

```

## Traverse Tree "Iterator"

I took the code for tree->list and reversed the order to make it highest to lowest and wrapped the second recursive call in a lambda so that I could halt traversal of the tree. This allowed me to avoid creating like a few redundant lists.

```

(define (traverse-left-tree tree)
  ;; reverse in-order traversal of binary-tree
  ;; the exact same thing as tree->list
  ;; except I reversed the order of recursion
  ;; of the branches to go from greatest to least
  ;; and put a lambda around the call to recurse on left
  ;; to allow for halted traversal
(define (iter tree result)
  (if (null? tree)
      result
      (iter
        (right-branch tree)
        (cons (entry tree)
          (lambda ()
            (iter
              (left-branch tree)
              result))))))
(iter tree nil))

```

## Union

If the sets we want to Union are represented in the form *(head . tail)* from greatest to least (as they are if we pass them to the *traverse-left-tree* function above)

```

(define set-1 ((max-entry set-1) (- set-1 (max-entry set-1))))
(define set-2 ((max-entry set-2) (- set-2 (max-entry set-2))))
.
.
.

(define set-n ((max-entry set-n) (- set-n (max-entry set-n))))

```

Then it can be shown that big-boy:

```
(define big-boy (apply max (map car (list set-1 set-2 .. set-n)) ))
```

Is not in any set-x where:

```
(not (= big-boy (car set-x)))
```

By definition (car set-x) is not equal to big-boy and because big-boy is the max of all heads then (car set-x) is smaller than big-boy. Therefore no other element in set-x is = big-boy because every other element of the set is also smaller than (car set-x) because greatest first order.

So we can safely recurse on

```

(union (cons big-boy result)
  (map
    ;; (cdr s) is wrapped in a lambda so we have to call it
    (lambda (s) (if (= (car s) big-boy) ((cdr s)) s))
    sets))

```

## Here is the actual solution

```

(define (union-set-tree . sets)
(define (iter result travelers)
  (if (null? travelers) ;; if we have no sets to work with return
      result
      (let ((next-entry;; add the largest entry found to the union
            (apply
              max
              (map car travelers))))
        (iter
          (cons next-entry
                result)
          (remove-nulls
            (map
              (lambda (trav)
                (if (or
                      (null? trav)
                      (not (= next-entry (car trav)))))
                    trav
                    ((cdr trav))))
              travelers))))))
  (list->tree (iter nil (remove-nulls (map traverse-left-tree sets))))))

```

## Intersection

The logic for set intersection is similar. We iterate over the trees represented as (head . tail) in the same way but instead of adding max automatically we only add it if each head equals max. This is because if each head does not equal max then all the heads that do not equal max are smaller than max so no other element in those sets can be max, and it is not in the intersection.

## Solution

```
(define (intersection-set-tree . sets)
  (define (iter result travelers)
    ;; if any of the sets are null
    ;; then the intersection is null
    ;; so return result
    (if (any? null? travelers)
        result
        (let ((new-max
              (apply;; apply is used to allow max to operate on a list
              max
              (map car travelers))))
            (let ((new-result
                  ;; if all heads equal max then add it to the result
                  (if (all?
                        (lambda (trav)
                          (= new-max (car trav)))
                        travelers)
                      (cons new-max result)
                      result)))
                (iter
                  ;; recurse with new result popping the max off each set
                  ;; we find it in
                  new-result
                  (map
                    (lambda (trav)
                      (if (or (null? trav)
                            (not (= new-max (car trav))))
                          trav
                          ((cdr trav)))
                      travelers))))))
          (let ((trees (map traverse-left-tree sets));;turns trees into (head . tail)
                (if (any? null? trees)
                    ;; null set intersected with any other set is the null set
                    nil
                    (list->tree (iter nil (map traverse-left-tree sets)))))))
```

---

[<< Previous exercise \(2.64\)](#) | [Index](#) | [Next exercise \(2.66\) >>](#)

---

# sicp-ex-2.66

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.65) | Index | Next exercise (2.67) >>

x3v binary trees are fast af

```
;; returns false if key not in records, else key
(define (lookup given-key set-of-records)
  (if (null? set-of-records) #f
      (let ((parent (entry set-of-records)))
        (cond ((eq? parent '()) #f)
              ((= given-key parent) parent)
              (else
                (lookup given-key
                      (if (< given-key parent)
                          (left-branch set-of-records)
                          (right-branch set-of-records)))))))
```

meteorgan

The solution is similar to element-of-set?

```
(define (lookup given-key set-of-records)
  (cond ((null? set-of-records) #f)
        ((= given-key (key (entry set-of-records)))
         (entry set-of-records))
        ((< given-key (key (entry set-of-records)))
         (lookup given-key (left-branch set-of-records)))
        (else (lookup given-key (right-branch set-of-records)))))
```

```
(define (lookup given-key set-of-records)
  (if (null? set-of-records)
    false
    (let ((node-entry (entry set-of-records)))
      (let ((node-key (key node-entry)))
        (if (= given-key node-key)
          node-entry
          (lookup given-key
            ((if (< given-key node-key) left-branch right-branch) set-of-
records)))))))
```

chessweb

A record looks like this: ((key value) left-branch right-branch)

```
(define (lookup given-key set-of-records)
  (cond
    ((null? set-of-records)
     #f)
    ((= given-key (entry (car (set-of-records))))
     (cadar set-of-records)) ; evaluates to value
    ((< given-key (entry (car (set-of-records))))
     (lookup given-key (left-branch set-of-records)))
    ((> given-key (entry (car (set-of-records))))
     (lookup given-key (right-branch set-of-records)))))
```

# sicp-ex-2.67

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

<< Previous exercise (2.66) | Index | Next exercise (2.68) >>

---

Define an encoding tree and a sample message:

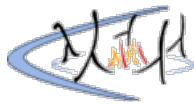
```
(define sample-tree
  (make-code-tree (make-leaf 'A 4)
                 (make-code-tree
                   (make-leaf 'B 2)
                   (make-code-tree (make-leaf 'D 1)
                                 (make-leaf 'C 1)))))

(define sample-message '(0 1 1 0 0 1 0 1 0 1 1 1 0))
```

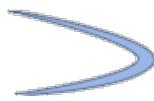
Use the decode procedure to decode the message, and give the result.

```
;; '(0 1 1 0 0 1 0 1 0 1 1 1 0)
;; '(A D     A B   B   C     A)
```

Last modified : 2022-08-27 08:54:54  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-2.68



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.67) | Index | Next exercise (2.69) >>

```
(define (encode message tree)
  (if (null? message)
      '()
      (append (encode-symbol (car message) tree)
              (encode (cdr message) tree)))))

(define (encode-symbol sym tree)
  (if (leaf? tree)
      (if (eq? sym (symbol-leaf tree))
          '()
          (error "missing symbol: ENCODE-SYMBOL" sym))
      (let ((left (left-branch tree)))
        (if (memq sym (symbols left))
            (cons 0 (encode-symbol sym left))
            (cons 1 (encode-symbol sym (right-branch tree)))))))
```

x3v

Simple recursive solution

```
(define (encode-symbol char tree)
  (cond ((leaf? tree) '())
        ((memq char (symbols (left-branch tree)))
         (cons 0 (encode-symbol char (left-branch tree))))
        ((memq char (symbols (right-branch tree)))
         (cons 1 (encode-symbol char (right-branch tree))))
        (else (error "symbol not in tree" char)))))

;; test
(equal? (encode (decode sample-message sample-tree) sample-tree) sample-message)
```

aQuaYi.com

I think my solution is clean.

```
(define (encode message tree)
  (if (null? message)
      '()
      (append (encode-symbol (car message) tree)
              (encode (cdr message) tree)))))

(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else (element-of-set? x (cdr set)))))

(define (encode-symbol symbol tree)
  (define (search symbol tree)
    (cond ((leaf? tree) '())
          ((element-of-set? symbol (symbols (left-branch tree)))
           (cons 0 (encode-symbol symbol (left-branch tree))))
          (else (cons 1 (encode-symbol symbol (right-branch tree)))))))
  (if (element-of-set? symbol (symbols tree))
      (search symbol tree)
      (error "try to encode NO exist symbol -- ENCODE-SYMBOL" symbol)))
```

```
;;
;; EXERCISE 2.68
;;

;; helpers
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else (element-of-set? x (cdr set)))))
```

```

(define (make-leaf symbol weight)
  (list 'leaf symbol weight))
(define (leaf? object)
  (eq? (car object) 'leaf))
(define (symbol-leaf x) (cadr x))
(define (left-branch tree) (car tree))
(define (right-branch tree) (cadr tree))
(define (symbols tree)
  (if (leaf? tree)
      (list (symbol-leaf tree))
      (caddr tree)))
(define (weight-leaf x) (caddr x))
(define (make-code-tree left right)
  (list left
        right
        (append (symbols left) (symbols right))
        (+ (weight left) (weight right))))
(define (weight tree)
  (if (leaf? tree)
      (weight-leaf tree)
      (caddr tree)))

(define (encode message tree)
  (if (null? message)
      '()
      (append (encode-symbol (car message) tree)
              (encode (cdr message) tree)))))

;; solution
(define (encode-symbol smb tree)
  (define (branch-correct? branch)
    (if (leaf? branch)
        (equal? smb (symbol-leaf branch))
        (element-of-set? smb (symbols branch)))))

  (let ((lb (left-branch tree))
        (rb (right-branch tree)))
    (cond ((branch-correct? lb)
           (if (leaf? lb) '(0) (cons 0 (encode-symbol smb lb))))
          ((branch-correct? rb)
           (if (leaf? rb) '(1) (cons 1 (encode-symbol smb rb))))
          (else (error "bad symbol -- ENCODE-SYMBOL" bit)))))

;; tests
(define sample-tree
  (make-code-tree (make-leaf 'A 4)
                 (make-code-tree
                   (make-leaf 'B 2)
                   (make-code-tree (make-leaf 'D 1)
                                 (make-leaf 'C 1)))))
  (encode '(A D A B B C A) sample-tree)
; (0 1 1 0 0 1 0 1 0 1 1 1 0)

```

ff0000

From a SICP newbie :-):

```

(define (encode message tree)
  (if (null? message)
      '()
      (append (encode-symbol (car message) tree)
              (encode (cdr message) tree)))))

(define (encode-symbol symbol tree)
  (define (encode-symbol-1 result t)
    (cond ((null? t) '())
          ((leaf? t)
           (if (not (eq? symbol (symbol-leaf t)))
               '()
               result))
          (else
            (let ((left-iter (encode-symbol-1 (cons 0 result)
                                              (left-branch t)))
                  (right-iter (encode-symbol-1 (cons 1 result)
                                              (right-branch t))))
                (cond ((and (not (null? left-iter))
                            (not (null? right-iter)))
                       (error "malformed tree -- ENCODE-SYMBOL" tree))
                      ((null? left-iter) right-iter)
                      ((null? right-iter) left-iter)
                      (else '())))))
    (let ((encoded-symbol (reverse (encode-symbol-1 '() tree))))
      (if (null? encoded-symbol)
          (error "symbol not found -- ENCODE-SYMBOL" symbol)
          encoded-symbol)))

(define sample-tree

```

```

(make-code-tree (make-leaf 'A 4)
  (make-code-tree
    (make-leaf 'B 2)
    (make-code-tree (make-leaf 'D 1)
      (make-leaf 'C 1)))))

(define sample-message '(0 1 1 0 0 1 0 1 0 1 1 0))

(define sample-word (decode sample-message sample-tree))

```

testing:

```

1 ]=> (equal? (encode sample-word sample-tree) sample-message)
;Value: #t

```

eliyak

```

(define (encode-symbol symbol tree)
  (cond
    ((leaf? tree) '())
    ((member symbol (symbols tree))
     (let ((left (left-branch tree)) (right (right-branch tree)))
       (if (member symbol (symbols left))
           (cons 0 (encode-symbol symbol left))
           (cons 1 (encode-symbol symbol right)))))
    (else (error "bad symbol -- ENCODE-SYMBOL" symbol))))

```

Siki

```

(define (encode-symbol symbol tree)
  (if (not (element-of-set? symbol (symbols tree)))
      (error "symbol cannot be encoded")
      (if (leaf? tree)
          '()
          (let ((left-set (symbols (left-branch tree)))
                (right-set (symbols (right-branch tree))))
            (cond ((element-of-set? symbol left-set)
                   (cons 0 (encode-symbol symbol (left-branch tree))))
                  ((element-of-set? symbol right-set)
                   (cons 1 (encode-symbol symbol (right-branch tree))))))))))

```

Kaucher

I had the same approach. You do the error check for all nodes you visit. Since a node knows which symbols its children contain, you only need to check it once.

```

(define (encode-symbol symbol tree)
  (define (search symbol tree)
    (let ((left-branch (left-branch tree))
          (right-branch (right-branch tree)))
      (cond ((leaf? tree) '())
            ((element-of-set? symbol (symbols left-branch))
             (cons 0 (search symbol left-branch)))
            ((element-of-set? symbol (symbols right-branch))
             (cons 1 (search symbol right-branch))))))
  (if (not (element-of-set? symbol (symbols tree)))
      (error "symbol not in tree -- ENCODE-SYMBOL" symbol)
      (search symbol tree)))

```

anonymous

Short and simple. Also iterative.

```

(define (encode-symbol symbol tree)
  (define (encode-symbol-1 tree bits)
    (cond ((and (leaf? tree) (eq? symbol (symbol-leaf tree)))
           (reverse bits))
          ((memq symbol (symbols (left-branch tree)))
           (encode-symbol-1 (left-branch tree) (cons 0 bits)))
          ((memq symbol (symbols (right-branch tree)))

```

```

  (encode-symbol-1 (right-branch tree) (cons 1 bits)))
  (else (error "symbol not in tree: ENCODE-SYMBOL" symbol))))
(encode-symbol-1 tree '())

```

Shubhro Has both iterative and recursive

```

(define (encode-symbol symbol tree)
  (define (encode-1 symbol tree result)
    (cond ((leaf? tree) result)
          ((element-of-set? symbol (symbols (left-branch tree)))
           (encode-1 symbol (left-branch tree) (append result '(0))))
          (else
            (encode-1 symbol (right-branch tree) (append result '(1))))))
  (define (encode-1-rec symbol tree)
    (cond ((leaf? tree) '())
          ((element-of-set? symbol (symbols (left-branch tree)))
           (cons 0 (encode-1-rec symbol (left-branch tree))))
          (else
            (cons 1 (encode-1-rec symbol (right-branch tree))))))

(if (not (element-of-set? symbol (symbols tree)))
  '()
  (encode-1-rec symbol tree)))

```

vz use a intree? function

```

(define (encode-symbol symbol branch)
  (define (in? symbol symbols)
    (if (null? symbols) false
        (if (eq? symbol (car symbols)) true
            (in? symbol (cdr symbols)))))
  (define (intree? symbol tree)
    (if (leaf? tree) (eq? symbol (symbol-leaf tree))
        (in? symbol (symbols tree))))
  (cond ((leaf? branch) '())
        ((intree? symbol (left-branch branch))
         (cons 0 (encode-symbol symbol (left-branch branch))))
        ((intree? symbol (right-branch branch))
         (cons 1 (encode-symbol symbol (right-branch branch))))
        (else (error "error"))))

```

GPE Use what we've learnt in chap 1 (procedural abstraction) to simplify the code

```

(define (encode-symbol symbol tree)
  (define (has-symbol side)
    (element-of-set? symbol (symbols (side tree))))
  (define (check-branch branch bit)
    (if (has-symbol branch)
        (cons bit (encode-symbol symbol (branch tree))))
  (cond ((leaf? tree) '())
        ((check-branch left-branch 0))
        ((check-branch right-branch 1))
        (else (error "no such symbol" symbol))))

```

Sphinxsky

Although it's a little different from the SICP, it should be more efficient.

```

;; Convert a Huffman coded binary tree into a list
(define (tree->list bits branch)
  (if (leaf? branch)
      (list (cons
              (symbol-leaf branch)
              (reverse bits)))
      (append
        (tree->list (cons 0 bits) (left-branch branch))

```

```

        (tree->list (cons 1 bits) (right-branch branch)))))

(define (encode message tree)

  (define symbol-list (tree->list '() tree))

  (define (encode-symbol symbol symbol-list)
    (let ((this-code
           (filter (lambda (x) (eq? (car x) symbol)) symbol-list)))
      (if (null? this-code)
          (error "bad symbol -- find-code" symbol)
          (cdar this-code)))))

  (define (encode-rec message symbol-list)
    (if (null? message)
        '()
        (append
          (encode-symbol (car message) symbol-list)
          (encode-rec (cdr message) symbol-list)))))

  (encode-rec message symbol-list))

```

caioB

Many examples before included a procedure to test if a symbol was contained in the tree or not, but this procedure was not define (e.g. GPE's has-symbol and eliyak's member)

Here is a solution using let to define a test to see if a symbol is contained in the tree. Luckily the way we are expressing trees already tells us which symbols each node contains. We can use that not only to create the error message, but also to decide if we want do go to the left or right branch:

```

;;
(define (encode-symbol symbol tree)
  (let ((ingroup? (lambda (symbol tree) (memq symbol (caddr tree)))))
    (if (not (ingroup? symbol tree))
        (error "bad symbol -- NOT ON TREE")
        (cond ((eq? symbol (symbol-leaf (left-branch tree))) '(0))
              ((eq? symbol (symbol-leaf (right-branch tree))) '(1))
              ((ingroup? symbol (right-branch tree)) (cons 1 (encode-symbol symbol
(right-branch tree)))
                  ((ingroup? symbol (left-branch tree)) (cons 0 (encode-symbol symbol (left-
branch tree))))))))

```

cody

```

;; Use an recursive function encode-symbol return the result or true or false
(define (encode-symbol s tree)
  (define (encode-symbol-i s current-tree result)
    [cond
      [(leaf? current-tree)
       (if (eq? (symbol-leaf current-tree) s)
           result
           false)]
      [else
       [let ([left-result
             (encode-symbol-i s (left-branch current-tree) (append result (list 0)))]
         (if left-result
             left-result
             [let ([right-result
                   (encode-symbol-i s (right-branch current-tree) (append result (list
1))])
               (if right-result
                   right-result
                   (error "no symbol s in the tree"))]]))]
     (encode-symbol-i s tree '())))

```

chessweb

Self contained solution

```

(define (encode message tree)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1) (append (cdr list1) list2))))

```

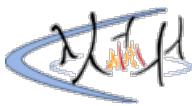


My solution works by tracing the path to the leaf node containing the target symbol recording the direction taken in the path with a 0 for left and 1 for right.

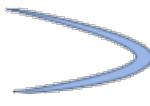
```
;; HELPER FUNCTIONS BEGIN
(define (fold-left op initial sequence)
  ; here
  (define (iter seq result)
    (if (null? seq)
        result
        (iter (cdr seq)
              (op (car seq)
                  result))))
  (iter sequence initial))

(define (any? predicate? elements)
  (fold-left
   (lambda (left right)
     (or (predicate? left) right))
   #f
   elements))
;; HELPER FUNCTIONS END

;;SOLUTION
(define (encode-symbol symbol tree)
  (define (in-symbols? tree)
    (any? (lambda (x) (eq? symbol x)) (symbols tree)))
  (cond
    ((and (leaf? tree) (eq? (symbol-leaf tree) symbol))
     '())
    ((in-symbols? (left-branch tree))
     (cons 0 (encode-symbol symbol (left-branch tree))))
    ((in-symbols? (right-branch tree))
     (cons 1 (encode-symbol symbol (right-branch tree))))
    (else (error "invalid symbol -- ENCODE-SYMBOL --" symbol))))
```



# sicp-ex-2.69



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

[<< Previous exercise \(2.68\)](#) | [Index](#) | [Next exercise \(2.70\) >>](#)

Exercise 2.69: The following procedure takes as its argument a list of symbol-frequency pairs (where no symbol appears in more than one pair) and generates a Huffman encoding tree according to the Huffman algorithm.

```
(define (generate-huffman-tree pairs)
  (successive-merge (make-leaf-set pairs)))
```

`make-leaf-set` is the procedure given above that transforms the list of pairs into an ordered set of leaves. `successive-merge` is the procedure you must write, using `make-codetree` to successively merge the smallest-weight elements of the set until there is only one element left, which is the desired Huffman tree. (This procedure is slightly tricky, but not really complicated. If you find yourself designing a complex procedure, then you are almost certainly doing something wrong. You can take significant advantage of the fact that we are using an ordered set representation.)

bro\_chenzox

There is a little modification to adjoin-set for result that makes from a pair set

```
' ((A 4) (B 2) (C 1) (D 1))
```

the following Huffman's tree

```
' ((leaf A 4) ((leaf B 2) ((leaf D 1) (leaf C 1) (D C) 2) (B D C) 4) (A B D C) 8) :
```

```
(define (adjoin-set x set)
  (cond ((null? set) (list x))
        ((> (weight x) (weight (car set))) (cons (car set)
                                                 (adjoin-set x (cdr set))))
        (else (cons x set))))
```

"This procedure is slightly tricky, but not really complicated. If you find yourself designing a complex procedure, then you are almost certainly doing something wrong." This is not our case, right? (I use embedded foldl instead of accumulate) :

```
(define (successive-merge set)
  (foldl make-code-tree (car set) (cdr set)))
```

jirf

I used the same strategy as chessweb. The text said "Successive-merge is the procedure you must write, using make-code-tree to successively merge the *smallest-weight elements*".

Although we can expect the initial input to be ordered, After merging the first code-tree that is no longer guaranteed. For example

```
(4 5 6 7 8 8.5) -merge-> (9 6 7 8 8.5)
-merge-> (15 7 8 8.5) ...
```

Easiest way I could think of successively merging the smallest elements was to order the set before each successive-merge, which can be done in O(n) time because the (intersection tree-set (make-set newly-created-tree)) is ordered.

## Solution

```
(define (generate-huffman-tree pairs)
  (define (order-tree-set tree tree-set)
    (cond ((null? tree-set) (list tree))
          ((>= (weight (car tree-set)) (weight tree))
           (cons tree tree-set))
          ((< (weight (car tree-set)) (weight tree))
           (cons tree tree-set))))
```

```

        (cons (car tree-set)
              (order-tree-set tree (cdr tree-set))))))
(define (successive-merge leaf-set)
  (cond ((null? leaf-set) nil)
        ((null? (cdr leaf-set)) (car leaf-set))
        (else
          (let ((new-tree (make-code-tree (cadr leaf-set)
                                         (car leaf-set))))
            (successive-merge (order-tree-set new-tree
                                             (cddr leaf-set)))))))
  (successive-merge (make-leaf-set pairs)))

```

```

(define (successive-merge trees)
  (let ((lightest-tree (car trees)) (heavier-trees (cdr trees)))
    (if (null? heavier-trees)
        lightest-tree
        (successive-merge (adjoin-set (make-code-tree lightest-tree (car heavier-trees))
                                      (cdr heavier-trees)))))))

```

brave one

@tyler's answer has too much abstraction sorry: `first-in-ordered-set`, `second-in-ordered-set`, `subset`.

since set operations are just inner workings of our program, it's perfectly fine (and i believe intended) to use direct list access operations here. abstraction is not a law, but rather guideline.

tyler

Chris's answer is correct, succinct, clear, but stylistically poor. Ordered sets may be implemented as lists, but abstraction dictates that we should never use list operators on them directly! By using a few name changes and methods we can respect the abstraction and remind ourselves what the objects actually are. I changed the code from Racket to Scheme for posting it here, sorry if there is some Racket remains in it.

```

(define (successive-merge tree-ordered-set)
  (if (= (size-of-set tree-ordered-set) 1)
      (first-in-ordered-set leaf-set)
      (let ((first (first-in-ordered-set tree-ordered-set))
            (second (second-in-ordered-set tree-ordered-set))
            (rest (subset tree-ordered-set 2)))
        (successive-merge (adjoin-set (make-code-tree first second)
                                     rest)))))

(define size-of-set length)
(define first-in-ordered-set car)
(define second-in-ordered-set cadr)
(define (subset set n)
  (if (= n 0)
      set
      (subset (cdr set) (- n 1))))

```

chris

vlad's answer is not correct!

Try this case:

```

(define test-tree (generate-huffman-tree '((A 3) (B 5) (C 6) (D 6))))
(encode '(A B C D) test-tree)

```

vlad's solution gets (1 1 1 1 0 0 1 0) while the shortest is just eight bits.

This is my solution. It's similar to frandibar's solution but I think there're some flaws in that solution.

```

(define (successive-merge leaf-set)
  (if (= (length leaf-set) 1)
      (car leaf-set)
      (let ((first (car leaf-set))
            (second (cadr leaf-set))
            (rest (cddr leaf-set)))
        (successive-merge (adjoin-set (make-code-tree first second)
                                     rest)))))


```

With the same test case, it gets (0 0 0 1 1 1 1 0).

frandibar

I guess this works...

```
(define (successive-merge leaf-set)
  (if (<= (length leaf-set) 1)
    leaf-set
    (let ([left (car leaf-set)]
          [right (cadr leaf-set)])
      (successive-merge (adjoin-set (make-code-tree left right) (cddr leaf-set))))))
```

vlad

Seems like the best solution is iterative ? Anyhow here is mine :

```
(define (generate-huffman-tree pairs)
  (successive-merge (make-leaf-set pairs)))

(define (successive-merge leaf-list)
  (define (successive-it ll tree)
    (if (null? ll) tree
        (successive-it (cdr ll) (make-code-tree (car ll) tree))))
  (successive-it (cdr leaf-list) (car leaf-list)))
```

anonymous

Both frandibar's and vlad's solutions are incorrect. The problem with frandibar's solution is that when the length of leaf-set is 1, it returns leaf-set. The correct solution returns (car leaf-set).

The correct solution can be more simply written as:

```
(define (successive-merge leaves)
  (if (null? (cdr leaves))
    (car leaves)
    (successive-merge
      (adjoin-set (make-code-tree (car leaves) (cadr leaves))
                  (cddr leaves)))))
```

ZelphirKaltstahl

I build on the knowledge form some of the provided solutions here and came up with the following one. Not much different from others, only using a let\* to assign names to some in between results.

```
(define (successive-merge ordered-nodes-set)
  (cond
    ;; For an empty set of nodes, we return the empty set.
    [(null? ordered-nodes-set) nil]
    ;; If there are less than 2 (1) elements in the set of nodes to merge, we are done.
    [(< (length ordered-nodes-set) 2) (car ordered-nodes-set)]
    ;; Otherwise merge the first two elements into a subtree, since they are the ones with
    ;; lowest weight.
    [else
      (let*
        ([new-node (combine-subtrees (car ordered-nodes-set)
                                      (cadr ordered-nodes-set))]
         [updated-ordered-nodes-set (adjoin-set new-node
                                              (cddr ordered-nodes-set))])
      (successive-merge updated-ordered-nodes-set)))]))
```

GPE

My solution

```
(define (successive-merge leafset)
  (if (null? (cddr leafset))
    leafset
    (successive-merge (cons (make-code-tree (car leafset) (cadr leafset)) (cddr leafset))))))
```

zenAndroid

Am i missing something here?

My solution is this ...

```
(define (generate-huffman-tree pairs)
  (define (succ-merge leafs)
    (cond ((null? leafs) (error "No leafs?"))
          (else (foldl make-huffman-tree (car leafs) (cdr leafs))))))
(succ-merge (make-leaf-set pairs)))
```

```
(generate-huffman-tree '((a 7) (b 8) (g 2) (t 6)))
((leaf b 8)
 ((leaf a 7) ((leaf t 6) (leaf g 2) (t g) 8) (a t g) 15)
 (b a t g)
 23)
> (generate-huffman-tree '((a 7)))
(leaf a 7)
> (generate-huffman-tree '())
<error symbols in racket> No leafs?
```

The most likely scenario is that i've missed something ??? but i dont know what ???

chessweb

This solution is inspired by the example on page 222 of the book. It has been tested with the following three examples as well as with excercise 2.70:

```
((C 1) (D 1))
((A 4) (B 2) (C 1) (D 1))
((A 8) (B 3) (C 1) (D 1) (E 1) (F 1) (G 1) (H 1))

(define (successive-merge leaf-set)
; inserts tree into tree-set such that the result remains
; ordered with respect to the weights
(define (insert tree tree-set)
  (cond
    ((null? tree-set) (list tree))
    ((< (weight tree) (weight (car tree-set))) (cons tree tree-set))
    (else (cons (car tree-set) (insert tree (cdr tree-set))))))
  (cond
    ((null? (cdr leaf-set)) (car leaf-set))
    (else (successive-merge (insert (make-code-tree (car leaf-set) (cadr leaf-set))
                                    (cddr leaf-set))))))
```

newone

The procedure `make-leaf-set` gives a leaf set as an ordered list and `adjoin-set` keeps this order untouched. In each expansion step, I just merge the first two leaves of the leaf set and insert the new node back to the set.

```
(define (generate-huffman-tree pairs)
  (successive-merge (make-leaf-set pairs)))

(define (successive-merge leafs)
  (cond ((null? leafs) '()) ; no leaf
        ((null? (cdr leafs)) (car leafs)) ; just one leaf
        (else ; two leaves or more
          (successive-merge (cddr
                            (adjoin-set
                              (make-code-tree (car leafs) (cadr leafs))
                              leafs))))))
```

# sicp-ex-2.70

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.69) | Index | Next exercise (2.71) >>

```
jirf
(define rock-lyrics
'((A 2)
  (BOOM 1)
  (GET 2)
  (JOB 2)
  (NA 16)
  (SHA 3)
  (YIP 9)
  (WAH 1)))
(define rock-tree
  (generate-huffman-tree rock-lyrics))

(define song
  '(GET A JOB
    SHA NA NA NA NA NA NA NA NA
    GET A JOB
    SHA NA NA NA NA NA NA NA
    WAH YIP YIP YIP YIP YIP YIP YIP YIP YIP
    SHA BOOM))

;; convenience func I wrote for printing a series of labeled messages
(display-block (cons "huffman-encoding length"
  (length (encode song rock-tree)))
  (cons "min fixed-length encoding length"
    (* (log (length rock-lyrics) 2) (length song))))
```

## ANSWER

```
huffman-encoding length
-----
84

min fixed-length encoding length
-----
108.0
```

pluies

My code gives:

```
(define rocktree (generate-huffman-tree '((A 2) (NA 16) (BOOM 1) (SHA 3) (GET
2) (YIP 9) (JOB 2) (WAH 1))))
rocktree

((leaf na 16) ((leaf yip 9) (((leaf a 2) ((leaf wah 1) (leaf boom 1) (wah boom) 2) (a wah
boom) 4) ((leaf sha 3) ((leaf job 2) (leaf get 2) (job get) 4) (sha job get) 7) (a wah boom
sha job get) 11) (yip a wah boom sha job get) 20) (na yip a wah boom sha job get) 36))
```

We can then encode the song:

```
(define rock-song '(Get a job Sha na na na na na na Get a job Sha na na na na na
na na Wah yip yip yip yip yip yip yip yip Sha boom))

(define encoded-rock-song (encode rock-song rocktree))

encoded-rock-song

(1 1 1 1 1 1 0 0 1 1 1 0 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 1 1 1 0 1 1 1
```

```
0 0 0 0 0 0 0 0 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 1 0 1 1 0 1 1
```

And compare the length of the encoded message vs. its fixed-length version:

```
(length encoded-rock-song)
84

; If we were to use a fixed-length encoding on that rock song, we would need 3 bits (8 =
2^3) per symbol, i.e.:
(* 3 (length rock-song))
108
```

A 22% gain in size seems to be coherent.

ZelphirKaltstahl

The book ignores newline characters printed in the text. I decided that they also should be encoded and came up with this:

```
(let*
  ([message
    (string-replace
      (string-append
        "Get a job\n"
        "Sha na na na na na na na\n"
        "Get a job\n"
        "Sha na na na na na na na\n"
        "Wah yip yip yip yip yip yip yip\n"
        "Sha boom") "\n" "\n")]
   [symbols-message (map string->symbol (string-split (string-upcase message) " "))]
   ; this list of count pairs would have to be calculated by the programm actually ...
   [huffman-tree (generate-huffman-tree (list
                                         (cons 'BOOM 1)
                                         (cons 'WAH 1)
                                         (cons 'A 2)
                                         (cons 'GET 2)
                                         (cons 'JOB 2)
                                         (cons 'SHA 3)
                                         (cons (string->symbol "\n") 5)
                                         (cons 'YIP 9)
                                         (cons 'NA 16)))))

  (display huffman-tree) (newline)
  (display (encode symbols-message huffman-tree)) (newline)
  (display (length (encode symbols-message huffman-tree)))) (newline))
```

# sicp-ex-2.71

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

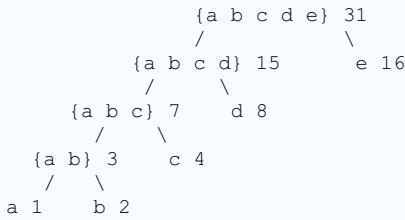
Search:

<< Previous exercise (2.70) | Index | Next exercise (2.72) >>

Exercise 2.71. Suppose we have a Huffman tree for an alphabet of  $n$  symbols, and that the relative frequencies of the symbols are  $1, 2, 4, \dots, 2^{n-1}$ . Sketch the tree for  $n=5$ ; for  $n=10$ . In such a tree (for general  $n$ ) how many bits are required to encode the most frequent symbol? the least frequent symbol?

NB: We will use letters from the alphabet to represent the symbols.

Example with  $n=5$ :



The tree for  $n=10$  looks similar, only larger.

The minimum number of bits to construct a symbol (i.e. the minimum depth to reach a leaf) for such trees is 1, for the symbol of weight  $2^{n-1}$ .

The maximum number of bits will be  $n-1$ , for the two symbols of least weight.

meteorgan

In this problem, the shapes of huffman trees are same. Every node's left/right branch only have one node. That's because  $1 + 2^1 + 2^2 + \dots + 2^{n-2} = 2^{n-1}-1 < 2^n-1$ . so every time we add a symbol into the Huffman tree, we make the symbol be the brother of the root node of the Huffman tree.

posxxa

For any  $n$ , the most frequent symbol code is 0. It's straightforward.

And the least frequent symbol, if  $n$  is infinite, we can simular merge process:

1, 2, 4, 8, 16, ...,  $n$

3, 4, 8, 16, ...,  $n$

7, 8, 16, ...,  $n$

We can discover the merge process is regulation and also straightforward. Imagining tree picture. The least frequent symbol code is (\*  $n$  1) (left corresponds 0, right corresponds 1.)

# sicp-ex-2.72

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.71) | Index | Next exercise (2.73) >>

Exercise 2.72. Consider the encoding procedure that you designed in exercise 2.68. What is the order of growth in the number of steps needed to encode a symbol? Be sure to include the number of steps needed to search the symbol list at each node encountered. To answer this question in general is difficult. Consider the special case where the relative frequencies of the  $n$  symbols are as described in exercise 2.71, and give the order of growth (as a function of  $n$ ) of the number of steps needed to encode the most frequent and least frequent symbols in the alphabet.

jirf

Major thanks to *RWitak*. I used his encode-symbol to improve my own which made answering this question require a lot less math.

## encode-symbol

```
(define (encode-symbol symbol tree)
  (cond ((leaf? tree)
         (if (eq? (symbol-leaf tree) symbol)
             '()
             (error "invalid symbol -- ENCODE-SYMBOL --" symbol)))
        ((memq symbol (symbols (left-branch tree)))
         (cons 0 (encode-symbol symbol (left-branch tree))))
        (else
         (cons 1 (encode-symbol symbol (right-branch tree))))))
```

## Solution

```
Best Case (symbol = most frequent) = O(1)
Worst Case = O(n)
```

## Explanation

The two operations that potentially grow in time complexity with the size of input are the memq operation for searching for the symbol in the left branch and the recursive call to search the rest of the tree.

## searching for the symbols in left-branch

```
((memq symbol (symbols (left-branch tree))))
```

Because as discussed in ex. 2.71 the number of symbols in the left branch is always 1 this operation does not grow in time-complexity with the growth in input size. Therefore it is  $O(1)$

## Tree Depth

The maximum depth of a huffman-tree with  $n$  nodes where their frequencies are:

```
(list 2^0 2^1 ... 2^(n-1))
```

Is  $n-1$ .

To see this lets look at how the huffman tree will be merged together. The  $n=1$  case is trivial. For each  $n > 1$  observe that the first two leaves have the same depth while each subsequent leaf merged into the tree at one "level" above (because we only ever merge two leaves together one time).

n=6

```

-> (1 2 4 8 16 32) <- max-depth = 1
-> ((1 2 3) 4 8 16 32) <- max-depth = 2
-> ((4 (1 2 3) 7) 8 16 32) <- max-depth = 3
-> ((8 (4 (1 2 3) 7) 15) 16 32) <- max-depth = 4
-> ((16 (8 (4 (1 2 3) 7) 15) 31) 32) <- max-depth = 5
-> ((32 (16 (8 (4 (1 2 3) 7) 15) 31) 63)) <- max-depth = 6
-> (32 (16 (8 (4 (1 2 3) 7) 15) 31) 63) <- max-depth = 5

```

## Best Case

In the function checks to see if the only symbol found in the left branch equals our target symbol. It will find it, recurse find that it is looking at a leaf with the correct symbol value and return. Therefore  $O(1)$

## Worst Case

The function will have to check if the symbol is a leaf/symbol in left branch (1) n times (n). So  $O(1 \cdot n) = O(n)$ .

RWitak

I think the answer to this question depends too much on the implementations to be answered generally. Take this version of **encode-symbol**:

```

(define (encode-symbol symbol tree)
  (define (iter-encode symbol tree result)
    (if (leaf? tree)
        result
        (if (memq symbol (symbols (left-branch tree)))
            (iter-encode symbol (left-branch tree)
                         (append result (list 0)))
            (iter-encode symbol (right-branch tree)
                         (append result (list 1))))))
    (if (memq symbol (symbols tree))
        (iter-encode symbol tree '())
        (error "bad symbol -- ENCODE-SYMBOL" symbol)))

```

The symbol lists, just as the tree itself, could be made so that they're ordered by decreasing weight, e.g. when implemented similar to SICP's "sample-tree" (and remember that we only discuss special cases similar to Ex. 2.71!):

```

(define power2-huffman
  (make-code-tree (make-leaf 'A 16)
                 (make-code-tree
                   (make-leaf 'B 8)
                   (make-code-tree
                     (make-leaf 'C 4)
                     (make-code-tree
                       (make-leaf 'D 2)
                       (make-leaf 'E 1))))))

```

Codes:

```

A 0
B 10
C 110
D 1110
E 1111

```

In this scenario, **encode-symbol** only checks the full list once. Then it iterates over the tree, only ever checking the left branch - which in the special case we're supposed to examine always has just a single leaf with a single symbol, no matter how big the tree.

This means that in more than half of all cases (16/31), we're done with one iteration -  $O(1)$ . In more than half of the remaining cases, we need 2 iterations, each checking just a single symbol. This continues until the last case, which is the rarest of all, and takes n iterations, making it  $O(n)$ . Also, the initial lookup is very fast for most cases, only the least likely case has to traverse the full list.

Now comes the speculative part, as I suck at math: Given that in my implementation, the number of cases that depend on deeper levels gets cut in half and each step on its own is  $O(n)$ , I estimate the overall complexity of that procedure as about  $O(\log n)$  - please correct me, if that's wrong.

Apart from all that, if we know beforehand that only a "powers-of-two" kind of tree will be used (and that it's implemented in the way given above), we don't even need the tree itself for encoding, just the ordered list of all symbols:

```

(define (power2-encode-symbol symbol power2-huffman)
  (define (traverse symlist)
    (cond ((null? symlist)
           (error "bad symbol -- POWER2-ENCODE-SYMBOL" symbol))
          ((eq? symbol (car symlist))
           (if (null? (cdr symlist))
               '(1)
               '(0)))
          (else (cons 1 (traverse (cdr symlist)))))))
  (traverse (symbols power2-huffman)))

```

In this very efficient encoding algorithm, the resulting code is already finished by the time we checked if the symbol is even contained in the tree! I guess it has the same overall complexity, but fewer steps in reality.

aos

Here's my take on this problem.

NB: This specifically refers to the *encode-symbol* procedure, which is a sub-procedure of the actual *encode*. If we were to extend the solution to *encode*, we'd have to multiply by an additional  $n$  for the entire message.

For the *encode-symbol* procedure in 2.68:

- Search the symbol list at each node:  $O(n)$  time
- Then take  $\log_n$  branches
- Total:  $O(n * \log_n)$

For the special case described in 2.71:

1. Encoding the most frequent symbol:

- Search through symbol list:  $O(n)$  time
- Take the first single branch, since it will be at the top of the list: constant
- Total:  $O(n)$

2. Encoding the least frequent symbol:

- Search through symbol list at each level:  $O(n)$  time
- Take the next branch, since we are only removing one node, it would be:  $O(n - 1)$
- Total:  $O(n * (n - 1))$ , or  $O(n^2)$

Sphinxsky

The above answer is not comprehensive enough.

For the *encode-symbol* procedure in 2.68:

Consider the maximum and minimum

- The minimum value is the case of a full binary tree  $O(n * \log_n)$
- The maximum value is like in 2.71  $O(n^2)$
- The final answer is between  $O(n * \log_n)$  and  $O(n^2)$ .

Big Old Duck

My encode-symbol for the special case have the growth rate:

Most frequent symbol:  $O(1)$  Least frequent symbol:  $O(n)$  Any symbol other than two above:  $O(n * \log(f))$ , where  $f$  is the frequency of that symbol. Can the growth be dependent on two variables though? IDK.

Why you may ask?

So, here is my encode-symbol:

(define (encode-symbol symbol tree)

```

(cond ((leaf? tree) '())
      ((element-of-set?
        symbol
        (symbols (left-branch tree)))
       (cons 0
             (encode-symbol symbol
                           (left-branch tree))))
      ((element-of-set?
        symbol
        (symbols (right-branch tree)))
       (cons 1
             (encode-symbol symbol
                           (right-branch tree)))))

```

```
(encode-symbol symbol
  (right-branch tree)))
(else (error "bad symbol: ENCODE-SYMBOL"
  symbol))))
```

I assume that the symbols will be sorted by their frequencies (or weights), in increasing order. So, steps for searching will be roughly  $\log(2, f)$ . So, number of steps:

$$T(n) = \log(f) + T(n-1)$$

$$\begin{aligned} &= 2 * \log(f) + T(n-2) \\ &= r * \log(f) + T(n-r) \end{aligned}$$

That will continue until we reach the point where  $n-r=\log(f)$ , that is the symbol is at the end of the list and will branch off to a leaf. So,  $T(n) = (n-\log(f)) * \log(f) + T(\log(f))$ .  $T(\log(f))$  will be  $O(1)$ , since we only have to search a leaf. Thus,  $T(n) = O(n * \log(f))$ . Hence proved (I wouldn't call this a proof though, a proof is more sophisticated).

partj

For the general case, for a tree with  $n$  symbols, the order of growth in the number of steps for encoding a given symbol depends upon the position of the symbol within the tree, i.e. which branches lead to its leaf when starting at the root, whether left or right, and how many, and the structure of the tree, i.e. the symbol lists encountered at every node along the way.

We can consider the best and worst cases along two different axes of consideration and extrapolate for cases in between. The symbol could be located right below the root of the tree as one of its direct branches or it could be farthest from the root at a maximum depth of  $n-1$ . Also, we may get lucky with our searches in the symbol list at each node encountered so that the symbol is found in  $\Theta(1)$  steps every time. Or we may have to search through all the symbols at every node and since the number of symbols at every node is some function of  $n$ , we may consider the cost of searching to be  $\Theta(n)$  at every node encountered. Here are the orders of growth from permuting on these scenarios:

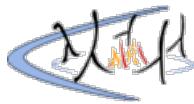
- best position, best search -  $\Theta(1)$
- best position, worst search -  $\Theta(n)$
- worst position, best search -  $\Theta(n)$
- worse position, worst search -  $\Theta(n^2)$

It is difficult to answer for the average case, since that would require us to know the structure of the average huffman tree. But if we consider that the average tree distributes its elements evenly between its branches (even if they aren't halves at every step), then the average element might very well be within a depth of  $\log n$  from the root of the tree (remember frequent symbols being closer to the tree helps our assumption). So our average complexity might be  $\Theta(n \cdot \log n)$ . If we can make our search more efficient so that it is  $\Theta(1)$  steps every time, our average complexity would improve to  $\Theta(\log n)$ .

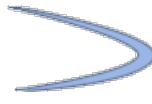
In the special case described in ex. 2.71, we observe that the depth of the tree is  $n-1$ .

For the most frequent symbol, we need only search once which would mean  $\Theta(n)$  in the worst case and  $\Theta(1)$  in the best.

For the least frequent symbol, we would encounter about  $n-1$  nodes along the way so depending on our search efficiency, we'd make searches of  $\Theta(n)$ ,  $\Theta(n-1)$  and so on, which would give us a number of steps of  $n*(n+1)/2$  or  $\Theta(n^2)$  order of growth in the worst case. Or with  $\Theta(1)$  efficient searching, we'd have  $\Theta(n)$  in the best case.



# sicp-ex-2.73



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.72) | Index | Next exercise (2.74) >>

jirf

Few notes.

The complex number example did not require that we call generic functions from inside the package. The package versions of deriv did require this because we needed to find the deriv of the data members (which could be any type of expression) in order to compute the answer.

The example package exported the data tagging to a func called tag outside of the internal package constructor. Did not do that in my packages because I needed to use the internal constructor in the internal deriv function.

Lastly, I assumed that the power in my exponential implementation was not the variable passed to deriv. This book is difficult enough already boyz, no need to deal with the chain rule.

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        (else ((get 'deriv (operator exp)) (operands exp)
                           var)))))

;; A. Above we have exported the type dispatch from the cond block internal to the
;; function to the assumed operation table.
;; We do not dispatch to the operation table for numbers or variables because both are
;; untagged. I suppose we could tag them but that would be a drag/why?

;; B.

(define (install-sum-package)
  (define (addend s)
    (cadr s))
  (define (augend s)
    (cadadr s))
  (define (deriv-sum s var)
    (make-sum (deriv (addend s) var);; deriv must be defined before this
              (deriv (augend s) var)))
  (define (make-sum x y)
    (list '+ x y))

;; interface to the rest of the system
(put 'deriv '+ deriv-sum)
(put 'make-sum '+ make-sum)
'done;; I guess this is just here for a print out message...

(define (make-sum add aug)
  ((get 'make-sum '+) add aug))

(define (install-product-package)
  (define (multiplier p)
    (cadr p))
  (define (multiplicand p)
    (cadadr p))
  (define (deriv-product p)
    (make-sum
      (make-product (deriv (multiplier p))
                    (multiplicand p))
      (make-product (multiplier p)
                    (deriv (multiplicand p))))))
  (define (make-product x y)
    (list '* x y))

;; interface to the rest of the system
(put 'deriv '(*) deriv-product)
(put 'make-product '*' make-product)
'done

;; C. Going to assume that exponent is not a func(var) as I do not feel like dealing with
;; the chain rule... like I really do not feel like dealing with that
(define (make-exponent-package)
  (define (base expression)
```

```

(cadr expression))
(define (power expression)
  (cadr expression))
(define (deriv-exponent expression)
  (cond
    ((= (power expression) 0) 0)
    ((= (power expression) 1) (base expression))
    (else
      (make-product
        (power expression)
        (make-exponent (base expression)
                      (- (power expression) 1))))))
(define (make-exponent base power)
  (list 'expt base power))

;; interface to the rest of the system
(put 'deriv '(expt) deriv-exponent)
(put 'make-exponent 'expt make-exponent)
'done)

;; D. Assuming that derivative system means the deriv generic function

(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        ;; line below swaps the position of the first and second arguments in the get func
        (else ((get (operator exp) 'deriv) (operands exp)
                           var))))
;; We would also have to swap the order of arguments to the put calls in the package
installation functions

```

```

;-----;
; EXERCISE 2.73
;-----

;; b
(define (install-sum-package)
  (define (make-sum a1 a2) (cons a1 a2))
  (define (addend s) (cadr s))
  (define (augend s) (caddr s))
  (define (deriv-sum s)
    (make-sum (deriv (addend s)) (deriv (augend s)))))

  (define (tag x) (attach-tag '+ x))
  (put 'deriv '(+) deriv-sum)
  (put 'make-sum '+
    (lambda (x y) (tag (make-sum x y))))
  'done)

(define (make-sum x y)
  ((get 'make-sum '+) x y))

(define (install-product-package)
  (define (make-product m1 m2) (cons m1 m2))
  (define (multiplier p) (cadr p))
  (define (multiplicand p) (caddr p))
  (define (deriv-product p)
    (make-sum
      (make-product (multiplier exp)
                    (deriv (multiplicand exp) var))
      (make-product (deriv (multiplier exp) var)
                    (multiplicand exp)))))

  (define (tag x) (attach-tag '.* x))
  (put 'deriv '(.*) deriv-product)
  (put 'make-product '.*
    (lambda (x y) (tag (make-product x y))))
  'done)

(define (make-product x y)
  ((get 'make-product '.*) x y))

(define (deriv x) (apply-generic 'deriv x))

```

BE

Here is my solution which can be tested in DrRacket.

```

#lang racket
(define *the-table* (make-hash));make THE table
(define (put key1 key2 value) (hash-set! *the-table* (list key1 key2) value));put
(define (get key1 key2) (hash-ref *the-table* (list key1 key2) #f));get

```

```

(define (install-symbolic-differentiation-package)

  (define (addend s) (car s))

  (define (augend s)
    (let ((cs (cdr s)))
      (if (null? (cdr cs))
          (car cs)
          (cons '+ cs)))))

  (define (make-sum a1 a2)
    (cond ((=number? a1 0) a2)
          ((=number? a2 0) a1)
          ((and (number? a1) (number? a2))
           (+ a1 a2))
          (else (list '+ a1 a2)))))

  (put 'deriv '+ (lambda (operands var)
                    (make-sum (deriv (addend operands) var)
                              (deriv (augend operands) var)))))

  (define (multiplier p) (car p))

  (define (multiplicand p)
    (let ((cs (cdr p)))
      (if (null? (cdr cs))
          (car cs)
          (cons '* cs)))))

  (define (make-product m1 m2)
    (cond ((or (=number? m1 0)
               (=number? m2 0))
           0)
           ((=number? m1 1) m2)
           ((=number? m2 1) m1)
           ((and (number? m1) (number? m2))
            (* m1 m2))
           (else (list '* m1 m2)))))

  (put 'deriv '*' (lambda (operands var)
                    (make-sum
                      (make-product
                        (multiplier operands)
                        (deriv (multiplicand operands) var))
                      (make-product
                        (deriv (multiplier operands) var)
                        (multiplicand operands))))))

  (define base car)
  (define exponent cadr)

  (define (make-exponentiation base exponent)
    (cond ((=number? base 0) 0)
          ((=number? exponent 1) base)
          ((=number? exponent 0) 1)
          ((and (number? base) (number? exponent))
           (expt base exponent))
          (else (list '** base exponent)))))

  (put 'deriv '** (lambda (operands var)
                    (make-product (exponent operands)
                                  (make-product (make-exponentiation (base operands)
                                                                     (- (exponent
                                                                     operands) 1))
                                              (deriv (base operands) var)))))

  'done)

  (define (variable? x) (symbol? x))

  (define (same-variable? v1 v2)
    (and (variable? v1)
         (variable? v2)
         (eq? v1 v2)))

  (define (=number? exp num)
    (and (number? exp) (= exp num)))

  (define (deriv exp var)
    (cond ((number? exp) 0)
          ((variable? exp)
           (if (same-variable? exp var)
               1
               0))
          (else ((get 'deriv (operator exp))
                  (operands exp)
                  var)))))


```

```

(define (operator exp) (car exp))
(define (operands exp) (cdr exp))

(install-symbolic-differentiation-package)
(deriv '(+ x x x) 'x)
(deriv '(* x x x) 'x)
(deriv '(+ x (* x (+ x (+ y 2)))) 'x)
(deriv '(** x 3) 'x)

```

meteorgan

Here is my answer.

```

;;a
number?, same-variable? are predicates. there's nothing to dispatch.

;; b
(define (install-sum-package)
  (define (sum-deriv expr var)
    (make-sum (deriv (addend expr) var)
              (deriv (augend expr) var)))
  (define (addend expr) (car expr))
  (define (augend expr) (cadr expr))
  (define (make-sum x1 x2)
    (cond ((and (number? x1) (number? x2)) (+ x1 x2))
          ((=number? x1 0) x2)
          ((=number? x2 0) x1)
          (else (list '+ x1 x2))))
  (define (mul-deriv expr var)
    (make-sum (make-product (multiplier expr)
                           (deriv (multiplicand expr) var))
              (make-product (multiplicand expr)
                           (deriv (multiplier expr) var))))
  (define (multiplier expr) (car expr))
  (define (multiplicand expr) (cadr expr))
  (define (make-product x1 x2)
    (cond ((and (number? x1) (number? x2)) (* x1 x2))
          ((=number? x1 1) x2)
          ((=number? x2 1) x1)
          ((or (=number? x1 0) (=number? x2 0)) 0)
          (else (list '* x1 x2))))
  (put 'deriv '+ sum-deriv)
  (put 'deriv '*' mul-deriv))

;; c
(define (exponentiation-deriv expr var)
  (make-product (exponent expr)
                (make-product
                  (make-exponentiation (base expr)
                                       (make-sum (exponent expr) -1))
                  (deriv (base expr) var))))
  (define (exponent expr)
    (cadr expr))
  (define (base expr)
    (car expr))
  (define (make-exponentiation base exponent)
    (cond ((=number? exponent 0) 1)
          ((=number? exponent 1) base)
          ((=number? base 1) 1)
          (else (list '** base exponent))))
  (put 'deriv '** exponentiation-deriv)

;;d
The only thing to do is changing the order of arguments in procedure "put".

```

brave one

I don't see the need for tags inside - operation itself is enough - and separation of functions'.

```

(load "deriv.scm")

(define (install-deriv-package)
  ;; internal procedures
  (define (deriv-sum pair var) ; pair as list
    (match pair
      [(list a b) ; a + b, yeah sorry do addend / augend
       (make-sum (deriv a var)
                 (deriv b var))]))
  (define (deriv-product pair var)

```

```

(match pair
  [(list a b) ; a * b
   (make-sum (make-product a
                           (deriv b var))
             (make-product b
                           (deriv a var))))]
(define (deriv-exponentiation pair var)
  (match pair
    [(list a b) ; a ^ b
     (make-product
       (make-exponentiation a
                             (make-sum b -1))
       (deriv a var))]))
;; interface to the rest of the system
(put 'deriv '+ deriv-sum)
(put 'deriv '*' deriv-product)
(put 'deriv '** deriv-exponentiation)
'done)

; just copy as is
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        (else ((get 'deriv (operator exp)) (operands exp)
                var))))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))

```

a0\_0x I'd like code can run. Below is my codes. Seperated my codes into 3 files, run 2\_73.scm to test the code

```

==> 2_73_b.scm <==
(define (install-sum-package)
  (define (addend s) (car s))
  (define (augend s) (cadr s))
  (define (make-sum a b)
    (cond
      ((eq? a 0) b)
      ((eq? b 0) a)
      ((and (number? a) (number? b)) (+ a b))
      (else (list '+ a b)))
    )
  )
  (define (deriv-sum s v)
    (make-sum (deriv (addend s) v) (deriv (augend s) v)))
  )
  (put 'deriv '+' deriv-sum)
'done)
(define (install-product-package)
  (define (multiplier s) (car s))
  (define (multiplicand s) (cadr s))
  (define (make-product a b)
    (cond
      ((or (eq? a 0) (eq? b 0)) 0)
      ((eq? a 1) b)
      ((eq? b 1) a)
      ((and (number? a) (number? b)) (* a b))
      (else '(* a b)))
    )
  )
  (define (make-sum a b)
    (cond
      ((eq? a 0) b)
      ((eq? b 0) a)
      ((and (number? a) (number? b)) (+ a b))
      (else (list '+ a b)))
    )
  )
  (define (deriv-product s v)
    (make-sum
      (make-product (deriv (multiplier s) v) (multiplicand s) )
      (make-product (multiplier s) (deriv (multiplicand s) v)))
    )
  )
  (put 'deriv '*' deriv-product)
'done)

==> 2_73.scm <==
(load "2_73_sys.scm")
(define (variable? x) (symbol? x))
(define (same-variable? x y) (and (variable? x) (variable? y) (eq? x y)))

```

```

(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (error "No method for these types -- APPLY-GENERIC"
                (list op type-tags)))
      )
    )
  )
)

(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        (else ((get 'deriv (operator exp)) (operands exp) var)))
  )
)

(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
;test
(load "2_73_b.scm")
(install-sum-package)
(install-product-package)
(deriv 'y 'x)
table
(get 'deriv '+)
(deriv '(+ (* x 3) (* y x)) 'x)
(deriv '(* 3 x) 'x)

==> 2_73_sys.scm <==
(define table (list ))
(define (put op type proc)
  (set! table (append table (list (list op type proc)))))
)
(define (get op type)
  (define (search op type t)
    (cond ((null? t) #f)
          ((and (eqv? (caar t) op) (eqv? (cadar t) type))
           (caddar t))
          (else (search op type (cdr t))))
    )
  )
  (search op type table)
)
(define (attach-tag type-tag contents)
  (cons type-tag contents)
)
(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "Bad tagged datum -- TYPE-TAG" datum)
  )
)
(define (content datum)
  (if (pair? datum)
      (cdr datum)
      (error "Bad tagged datum -- CONTENTS" datum)
  )
)
;(install-sum-package)
;(install-product-package)
;table

```

masque

I have found a deadly bug in almost every answer posted here before. It's about name conflict. For example in sum-package, in function deriv-sum, we are going to use make-sum again to producing the derivative. This make-sum, which we may intended to refer to the outer global make-sum defined by extracting from the table, is actually referring the inner local make-sum. This leads to the result derivative no longer a valid expression, which is just a pair without the type-tag. And the same problem appears in other packages too. My solution is to change the inner make-sum to make-sum-inner to avoid name conflict. Then we only have to change the make-sum to make-sum-inner in the (put 'make-sum '+ ...) and the program becomes correct. And I wonder if there are better solutions...

jirf

I noticed this as well. There should be no problem if we tag the return value of make-<expression type> in the body of the package version. We did that in 2.3.2 so if you just copy and paste make-<expression type> over from your old code you should be good.

tiendo1011

@masque We don't need tag here.

To find the operation to applied, we need to know the operation & the type.

Here the operation is the deriv, and the type is the operator (+/\*), which is easily extracted from the exp.

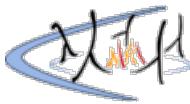
For the complex package example, the operation is provided, but the type is not, and there is no easy way to find the type of a complex number (hence the need to use tag).

tch

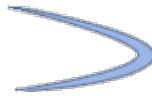
if your environment do not have a built-in put and get(like #lang sicp in Racket), just see Section 3.3.

```
(define (make-table) (list '*table*))
; 2d table
(define (lookup2 key-1 key-2 table)
  (let ((subtable (assoc key-1 (cdr table))))
    (if subtable
        (let ((record (assoc key-2 (cdr subtable))))
          (if record
              (cdr record)
              #f)
        )
      #f
    )
  )
(define (insert2! key-1 key-2 value table)
  (let ((subtable (assoc key-1 (cdr table))))
    (if subtable
        ; subtable exist
        (let ((record (assoc key-2 (cdr subtable))))
          (if record
              (set-cdr! record value) ; modify record
              (set-cdr! subtable
                (cons (cons key-2 value) (cdr subtable))) ; add
            record
          )
        )
      ; subtable doesn't exist, insert a subtable
      (set-cdr! table
        (cons (list key-1 (cons key-2 value)) ; inner subtable
          (cdr table)))
    )
  )
)

; put and get
(define *table* (make-table)) ; a global table
(define (put op type item)
  (insert2! op type item *table*)
)
(define (get op type)
  (lookup2 op type *table*)
)
```



# sicp-ex-2.74



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.73) | Index | Next exercise (2.75) >>

vf

;; SICP 2.74

```
;a. Implement get-record, based on a foo x division table.  
;  Each division file must have a division func to extract  
;  'type' and provide a installation package to include  
;  specific get-record. The output is a tagged record.  
;  Get-record must return false if doesn't find employee record (c).  
(define (attach-tag type-tag content) (cons type-tag content))  
(define (get-record employee-id file)  
  (attach-tag (division file)  
    ((get 'get-record (division file)) employee-id file)))  
  
;b. get-salary  
(define (get-salary record)  
  (let ((record-type (car record))  
        (record-content (cdr record)))  
    ((get 'get-salary record-type) record-content)))  
  
;c. find-employee-record  
(define (find-employee-record employee-id file-list)  
  (if (null? file-list)  
      #f  
      (let ((current-file (car file-list)))  
        (if (get-record employee-id current-file)  
            (get-record employee-id current-file)  
            (find-employee-record employee-id (cdr file-list))))))  
  
;d. New company must provide a installation package for its  
;  record-file as new division. This instalation must include  
;  the new division get-record and get-salary implementations.
```

palatin

Let's assume there is table indexed by Division name providing the 'record' and 'salary' generic procedures of one argument, the employee name and the record respectively. Here is a possible answer:

```
; ; a)  
(define (get-record division employee-name)  
  ((get division 'record) employee-name))  
  
; ; b)  
(define (get-salary division record)  
  ((get division 'salary) record))  
  
; ; c)  
(define (find-employee-record employee-name division-list)  
  (if (null? division-list)  
      #f  
      (or (get-record (car division-list) employee-name)  
          (find-employee-record employee-name (cdr division-list)))))
```

d) the new company needs to install its 'record' and 'salary' generic procedures into the lookup table using its name as the first key.

Siki

The answer above is slightly different from what I think. The table for this question should have a horizontal axis of division name(type) and a vertical axis of procedures(op).

Therefore: a) Each division's records consist of a single file, which contains a set of records keyed on employees' names. This single file should be structured as a list of employees' records, with a tag of the division name. And each employee's record should have a tag of their name. The way these tags are put must be supplied. E.g. the file of division-1 can be '(division-1 (Mike 2000) (Jack 3500)) Here is the corresponding answer:

```
(define (install-division-1-package)
```

```

;;internal procedures
(define (get-record name file)
  (cond ((null? file) (error "no result"))
        ((eq? name (get-name (cadr file))) (cons (cadr file)
                                                (get-record name (cdr file))))
        (else (get-record name (cdr file)))))

(define (get-name record)
  (car record))

;;interface to the rest of the system
(put 'get-record 'division-1 get-record)
(put 'get-name 'division-1 get-name)
'done)

(define (get-record name file)
  (apply-generic 'get-record name file))

(define (apply-generic op name file)
  (let ((division-name (type-tag file)))
    (let ((proc (get op division-name)))
      (if proc
          (proc name file)
          (error "no result")))))

(define (type-tag file)
  (car file))

```

b) Take division-1 above as an example, the answer just needs to be modified by adding a few lines:

```

;; Addition to the division-1 package
(define (get-salary name file)
  (cond ((null? file) (error "no result"))
        ((eq? name (get-name (cadr file))) (cons (cadr (cadr file))
                                                (get-salary name (cdr file))))
        (else (get-salary name (cdr file)))))

(put 'get-salary 'division-1 get-salary)

;;Addition to the environment
(define (get-salary name file)
  (apply-generic 'get-salary name file))

```

c) Just implement "get-record" to all the divisions.

```

(define (find-employee-record name list)
  (if (null? list)
      (error "no result")
      (append (get-record (car list))
              (find-employee-record name (cdr list)))))


```

d) Install a new package which specifies the procedures to look up name/salary in the new company's file.

sockyfeet

a) I am assuming that the individual divisions' files are structured such that there already exists, for each division, a get-record method. I am assuming there exists an (<operator>, personnel-file) table. The reason I am not assuming there exists an (<operator>, division) table is that this assumption would then require the additional assumption that there exists a method that takes personnel-file as argument and returns the division of that personnel-file. Since each division has exactly one personnel-file, we have a 1-1 correspondence between the two pieces of data. Indexing by personnel-file allows me to make one less assumption and have the same amount of columns. Thus, we have:

```

(define (get-record employee personnel-file)
  ((get 'get-record personnel-file) employee))

```

b) Now, I just assume that each record is structured such that each division can implement and has implemented their own get-salary method, that takes as argument the employee's record and returns the employee's salary. Thus:

```

(define (get-salary employee personnel-file)
  ((get 'get-salary personnel-file) (get-record employee personnel-file)))

```

c) I'm assuming that each individual divisions get-record method returns '()' if the employee is not employed in that division, and so the method that we implemented in part a) of this question returns '()' in that case, too. We have:

```

(define (find-employee-record employee division-files)
  (cond ((null? division-files) '()))

```

```
((not (null? (get-record employee (car division-files)))  
  (get-record employee (car division-files)))  
 (else (find-employee-record employee (cdr division-files)))))
```

- d) Call the newly acquired division-file new-division-file. Entries corresponding to (get-record, new-division-file) and (get-salary, new-division-file) need to be added to our (<operator>, division-file) table. That is, the newly acquired company needs to implement locally their own versions of get-record, get-salary, then we need to add these implementations to our table, and then the methods defined in a), b), c) will work.

# sicp-ex-2.75

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

<< Previous exercise (2.74) | Index | Next exercise (2.76) >>

---

meteorgan

It's similar to make-from-real-imag.

```
(define (make-from-mag-ang r a)
  (define (dispatch op)
    (cond ((eq? op 'real-part) (* r (cos a)))
          ((eq? op 'imag-part) (* r (sin a)))
          ((eq? op 'magnitude) r)
          ((eq? op 'angle) a)
          (else (error "Unknown op --- MAKE-FROM-MAG-ANG" op))))
```

---

Last modified : 2011-08-23 05:08:20  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-2.76

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

Regarding additivity discussed in this chapter:

- message-passing allows to add new types without changing already written code
- explicit dispatch allows to add new operations in the same manner
- data-directed approach allows us to add new types with corresponding operations and new operations for existing types as well just by adding new entries in the dispatch table

This is the well-known **expression problem**.

[\*\*<< Previous exercise \(2.75\) | Index | Next exercise \(2.77\) >>\*\*](#)

jirf

## System Changes For Adding Types/Operations

### Generic Operations with Direct Dispatch

#### Adding New Types

To add new types one must create the constructor/selectors and update the cond/if blocks which handle dispatch. You have to jump from operation to operation to update the dispatch tables but only one constructor is altered (created).

#### Adding new Operations

Creating new operations does not require any code changes outside of the creation of the new operation. Might have to jump around to make sure you include each data type in the dispatch table.

### Data-Directed Style

#### Adding New Types

Create and run an "installer" function. Installer should *a.* implement the constructor, selectors, and each operation *b.* install with the get procedure the function to the dispatch table.

This style only requires editing/calling the installer which can be done only editing one code location (no jumping :)). Although you might have to jump to your generic function section to remember all of the operations to implement.

#### Adding New Operations

Each operation dispatch case is written in a separate "installer" function. Adding a new operation involves creating a new table "row" (generic function) and each "column" (operation defined for a specific data type in installer). You have to create the new generic function, and edit each data type installer which requires the new operation. Lot of jumping :)

### Message-Passing Style

#### Adding New Types

Message-passing style internalizes both data and operations completely. Creating a new data type requires only the creation of the constructor. The resulting object must implement internally all selectors,

and operations. It is possible to add new types only editing one code location (no jumping).

## Adding New Operations.

Because each operation implementation is internal to the object adding new operations requires editing each data type constructor which requires the new operation.

## Recommendation: Frequent Type Additions

Message-passing and data-directed styles are equivalent optimal choices. Both internalize their operation implementations. Where operation additions grows slowly relative to type additions on average programmers will have to edit less code locations than direct-dispatch. In my experience this introduces less bugs/mistakes.

## Recommendation: Frequent Operation Additions

Direct-dispatch localizes operation logic internal to its definition for all data types. If data types are not constantly being added the code blocks which dispatch on type will not endlessly grow, which can be hard to manage.

brave one

Actually I don't see the difference: in both cases we have type as a unit, which contains operations. Just dispatch in data-driven variant is external and in message-passing internal. So types are easy to add always, new unit doesn't touch old stuff. And operations touch everything.

torinmr

Hmmm, people seem to be forgetting that there are three strategies here, not just two:

For generic operations with explicit dispatch (which is the same as the "functional programming approach" described in the wiki link above), adding new operations is easy (i.e. can be done without changing existing code).

For message passing, adding new types can be done without changing existing code, but adding new operations requires changing all the old code.

Data directed programming actually "solves" the Expression problem by allowing new types \*and\* new operations to be added without changing existing code: To add a new type I just fill out a new column in the table of operations, to add a new operation I fill out a new row. (Note that style used in the book of putting all the operations for e.g. the rectangular representation together makes it seem like adding a new operation to all the representations would be hard, but there's nothing stopping me from having my own define block where I register a 'complex-conjugate' method under 'rectangular' and 'polar'.)

Of course, the amount of code that needs to be written to add a new type or operation is about the same under all three approaches - the only difference is how spread out that code needs to be, and how easy it is to locate all the places where code needs to be changed. Data directed programming seems to optimize for the former at the expense of the latter.

krubar

I found torinmr answer to be correct one. What brave one I think is missing is that in data-directed approach one doesn't necessarily need to modify existing `install-package` declaration to add new operations. It can be done in newly written `install-package` block or just by registering new operations with series of `put` calls: either for new type or in the case we want to add operation for all of the types.

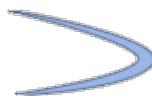
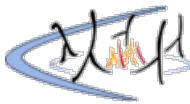
rwtak

I think Brian Harvey said it best in his 2011 Berkeley course on the topic, when he stated that "**old code should not stop working when you add new code!**"

In this case:

- Message-passing cannot equip old data types with new functionality *without changing the definitions in the existing type* - new types are no problem to existing code, as long as they implement all functions.
- With explicit dispatch every function includes the existing types and only works for them. When you add a new type, you have no way of including them in existing procedures without changing them.
- The lookup table of data directed programming allows for both types and procedures to be added quite seamlessly - even both at the same time! *No old code has to be changed, even if a new function involves an old type or vice versa.*





<< Previous exercise (2.76) | Index | Next exercise (2.78) >>

jirf

## Data Structure

Our complex number implementation constructors first call the polar or rectangular constructors, and tag the result with the 'complex tag resulting in the representation found in Figure 2.24.

```
('complex . ('rectangular . (x . y)))
('complex . ('polar . (r . a)))
```

When Mr. Reasoner called the magnitude function the type-tags call used in apply-generic returned the outer tag of z (complex). Because Mr. Reasoner had not put a magnitude function under the complex tag anywhere in his code this returns an error.

```
(define (apply-generic op . args)
  ;; as seen in 2.4.3
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags))) ;; <- here is where Louis gets in trouble
      (if proc
          (apply proc (map contents args))
          (error
            "no method for these types -- APPLY-GENERIC"
            (list op type-tags)))))

(define (install-polar-package)
  (define (magnitude z) (car z))
  ...
  (put 'magnitude '(polar) magnitude)
  ...
  'done)

(define install-rectangular-package)
...
(define (magnitude z)
  (sqrt (+ (square (real-part z)) (imag-part z)))))
...
(put 'magnitude 'rectangular magnitude)
...
'done)

(define (magnitude z)
  (apply-generic 'magnitude z))
```

After Louis implements Mrs. Hacker's suggestion to his complex package the apply-generic function will find the magnitude function under the complex tag, and call the magnitude function (again) on the "contents" of the complex number. Which, because z is the object found in Fig. 2.24, is ('rectangular . (3 . 4)). Magnitude will again invoke apply-generic. This time the magnitude function internal to the rectangular-package will be called on (3 . 4).

Apply-generic will be called 2 times and the rectangular-package magnitude function is what is finally called.

eric4brs

From exercise: "Louis Reasoner tries to evaluate the expression (magnitude z) where z is the object shown in figure 2.24. To his surprise, instead of the answer 5 he gets an error message from apply-generic, saying there is no method for the operation magnitude on the types (complex). He shows this interaction to Alyssa P. Hacker, who says ``The problem is that the complex-number selectors were never defined for complex numbers, just for polar and rectangular numbers. All you have to do to make this work is add the following to the complex package:'''"

```
(put 'real-part '(complex) real-part)
(put 'imag-part '(complex) imag-part)
(put 'magnitude '(complex) magnitude)
(put 'angle '(complex) angle)
```

This is acting as a pass-through. If we were only building a complex arithmetic system this would be adding a pointless extra level. But this extra switch is what allows us to combine complex with regular and rational numbers.

```
; From text:
; ----->|.----->|.----->|.|.
;           |           |           |
;           V           V           V V
;           complex     rectangular  3 4

; Figure 2.24: Representation of 3 + 4i in rectangular form.

; and also from text:

(define (real-part z) (apply-generic 'real-part z))
(define (imag-part z) (apply-generic 'imag-part z))
(define (magnitude z) (apply-generic 'magnitude z))
(define (angle z) (apply-generic 'angle z))
```

Also from exercise: "Describe in detail why this works. As an example, trace through all the procedures called in evaluating the expression (magnitude z) where z is the object shown in figure 2.24. In particular, how many times is apply-generic invoked? What procedure is dispatched to in each case?"

```
; Parse error: Closing paren missing.
;*** Use substitution rule:
(magnitude z)

;** First apply-generic:
(apply-generic 'magnitude z) ; where z is the whole objec including symbol 'complex.
;recall
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (error
            "No method for these types -- APPLY-GENERIC"
            (list op type-tags)))))
; substitution
(let ((type-tags '(complex)) ...))
(let ((proc (get op '(complex)))) ...)
(let ((proc magnitude) ...))
(if proc... ; true
  (apply magnitude (contents z)))
  (magnitude z-prime) ; where z-prime is the contents (the cdr) of the original object, that
  is, with the 'complex stripped off.

;** Second apply-generic:
(let ((type-tags '(rectangular)) ...))
(let ((proc (get op '(rectangular)))) ...)
(let ((proc (get 'magnitude '(rectangular))) ...))
(let ((proc (lambda (z) (sqrt (+ (square (real-part z))
                                    (square (imag-part z)))))) ...))

(if proc... ; true
  (apply (lambda (z) (sqrt (+ (square (real-part z))
                            (square (imag-part z)))))) (contents z-prime))
  (sqrt (+ (square 3) (square 4)))
5
```

meteorgan

Before Alyssa added the code, there is no magnitude operation for complex, hence the error. apply-generic is invoked twice, first dispatch is magnitude of complex, second is magnitude of rectangular.

ZelphirKaltstahl

There is some mistake here. There is no procedure magnitude for the 'complex package, so the first thing failing would be:

```
(put 'magnitude '(complex) magnitude)
```

In the code of the book the package does not contain such a procedure magnitude. It seems to assume that Alyssa also writes that code, but in the exercise 2.77 does not say so. The book also does not state in which package the code should be added. The specific procedures magnitude inside the packages rectangular and polar do not work for complex, because they assume, that they are working with either representation and are not aware of the additional tag 'complex attached to the data. So they would extract the wrong pieces from the data and then

fail. The implementation of magnitude inside the complex package would probably have to unwrap the tagged data by one tag, the complex tag, and perform a lookup in the table of operations itself:

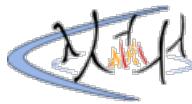
```
(define (magnitude z) (get 'magnitude (cadr z) (cdr z)))
;; get operation for magnitude of the data wrapped inside the data.
;; (cadr z) would become 'rectangular
;; (cdr z) would become (list 'rectangular (cons 3 4))
```

So basically with the information given in the book the tracing (as demanded in the exercise) of the calls would fail (correct me if I am wrong about this please). However, under reasonable assumptions, like a defined magnitude in the package for complex, I think the answer of meteorgan is correct.

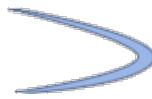
Hoonji

Perhaps this is what you're looking for. The generic magnitude selector is defined in **section 2.4.3**:

```
(define (magnitude z) (apply-generic 'magnitude z))
```



# sicp-ex-2.78



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.77) | Index | Next exercise (2.79) >>

```
;; -----
;; EXERCISE 2.78
;; -----  
  
(define (attach-tag type-tag contents)  
  (if (number? contents)  
      contents  
      (cons type-tag contents)))  
  
(define (type-tag datum)  
  (cond ((number? datum) 'scheme-number)  
        ((pair? datum) (car datum))  
        (else (error "Wrong datum -- TYPE-TAG" datum))))  
  
(define (contents datum)  
  (cond ((number? datum) datum)  
        ((pair? datum) (cdr datum))  
        (else (error "Wrong datum -- CONTENTS" datum))))
```

Siki

The answer above doesn't follow the requirement in this exercise. The right answer should be:

```
(define (type-tag datum)  
  (cond ((number? datum) datum)  
        ((pair? datum) (car datum))  
        (else (error "Wrong datum -- TYPE-TAG" datum))))  
  
(define (contents datum)  
  (cond ((number? datum) datum)  
        ((pair? datum) (cadr datum))  
        (else (error "Wrong datum -- CONTENTS" datum))))  
  
(define (attach-tag tag content)  
  (cond ((number? content) content)  
        ((pair? content) (cons tag content))))
```

The above correction is wrong. The original answer is correct.

The differences in and problems with the correction are as follows:

- in type-tag, return the raw datum when it is a number. This will not work once apply-generic is called because there is no entry in the operation lookup table for each number. Returning 'scheme-number' is the correct thing to do here.
- in contents, returning the cadr of the datum when it is a pair. Maybe just a typo? The original version (in the book) uses cdr; there's no reason this should change, because type tags are still attached to pairs with (cons tag content).

torinmr

Hmmm, I believe that the top version does have a bug, but it's not the one mentioned by Siki. The bug is that the condition in attach-tag should be

```
(eq? type-tag 'scheme-number)
```

Instead of

```
(number? contents)
```

This is because it's possible to create a tagged data type where the contents is just a number, but the type-tag is not scheme-number. (For example, suppose we implemented a "logarithm" type for storing very large numbers, where the stored value was the natural logarithm of the number we were representing.)

noname

If the above is correct, then the book itself is wrong, as it says explicitly to represent native numbers without the 'scheme-number' symbol.

atomik

I agree with torinmr. Checking the type-tag argument passed to attach-tag eliminates any ambiguity about what type the user wants the contents of their datum to have. Native numbers can still be represented if the attach-tag function simply returns the contents parameter like so:

```
(define (attach-tag type-tag contents)
  (if (eq? type-tag 'scheme-number)
      contents
      (cons type-tag contents)))
```

This does not violate the instructions given in the book which say "ordinary numbers should be represented simply as Scheme numbers rather than as pairs whose car is the symbol 'scheme-number'"

Too elaborate on the example torinmr gave, if the user wants to run

```
(attach-tag 'log 5)
```

the version given in this comment will return '(log 5). The implementations given at the top of the page will return 5, thus breaking any operations which expect data tagged with the 'log symbol. If the user runs

```
(attach-tag 'scheme-number 5)
```

the version given in this comment will return 5, just like the implementations given at the top of the page.

RWitak

After much confusion on my part, I also agree that torinmr's solution is optimal for additivity. The number itself is never represented as anything other than itself, only when deciding on how to tag and how to look up the number are there 'scheme-number' tags involved - which I think is acceptable as long as the number representation stays 5 instead of (**scheme-number . 5**).

jirf

Weighing in on the great *number?* debate. The purpose of this exercise is to make the system "work as before except that ordinary numbers should be represented simply as Scheme numbers rather than as pairs whose car is the symbol scheme-number".

To do this all we have to do is create a type dispatch in our type system that recognizes when the object is a number and returns the 'scheme-number' tag that was never there in order to route the arithmetic procedures (add, sub, div, mul) to the correct procedures.

```
(define (attach-tag t x)
  (if (number? x) x (cons t x)))
(define (type-content x)
  (cond
    ((pair? x)
     (if (symbol? (car x)) x
         (error "Invalid Type Tag. Tag Must Be Symbol Not " (car x)))
     ((number? x) (cons 'scheme-number x))
     (else (error "Unrecognizable Type For" x))))
  )
(define (type-tag x)
  (car (type-content x)))
(define (contents x)
  (cdr (type-content x)))
```

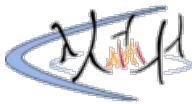
partj

In addition to the changes to the type-tag system, we can also now simplify the scheme-number package definitions (because we don't need to tag a scheme number anymore):

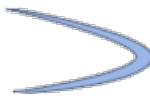
```
(define (install-scheme-number-package)
  (put 'add '(scheme-number scheme-number) +)
  (put 'sub '(scheme-number scheme-number) -)
  (put 'mul '(scheme-number scheme-number) *)
  (put 'div '(scheme-number scheme-number) /)
  'done)
```

The only thing required is that querying (type-tag some-scheme-number) should return 'scheme-number'.





# sicp-ex-2.79



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.78) | Index | Next exercise (2.80) >>

```
;; -----
;; EXERCISE 2.79
;; -----  
  
(define (install-scheme-number-package)
  ;; ...
  (put 'equ? '(scheme-number scheme-number) =)
  'done)  
  
(define (install-rational-package)
  ;; ...
  (define (equ? x y)
    (= (* (numer x) (denom y)) (* (numer y) (denom x))))
  ;; ...
  (put 'equ? '(rational rational) equ?))
  'done)  
  
(define (install-complex-package)
  ;; ...
  (define (equ? x y)
    (and (= (real-part x) (real-part y)) (= (imag-part x) (imag-part y))))
  ;; ...
  (put 'equ? '(complex complex) equ?))
  'done)  
  
(define (equ? x y) (apply-generic 'equ? x y))
```

I think it's best to define equ? in each implementation of complex:

```
(define (install-rectangular-package)
  ;; ...
  (put 'equ? '(rectangular rectangular)
    (lambda (x y) (and (= (real-part x) (real-part y))
                        (= (imag-part x) (imag-part y)))))
  'done)  
  
(define (install-polar-package)
  ;; ...
  (put 'equ? '(polar polar)
    (lambda (x y) (and (= (magnitude x) (magnitude y))
                        (= (angle x) (angle y)))))
  'done)  
  
(define (equ? x y) (apply-generic 'equ? x y))  
  
(define (install-complex-packages)
  ;; ...
  (put 'equ? '(complex complex) equ?))
  'done)
```

The above solution for defining equality for polar terms is incorrect. Two polar numbers are equal when they have the same magnitude and have angles that are congruent modulo  $2\pi$  (where the angles are measured in radians). Additionally, if the magnitude of a polar form is 0, it is equal to all other polar values with magnitude 0, regardless of their angle (i.e.  $r=0, \theta=10$  is equivalent to  $r=0, \theta=0$ ).

Another issue with this second solution is now you need to consider 4 cases:

- (1) `(rectangular rectangular)
- (2) `(polar rectangular)
- (3) `(rectangular polar)
- (4) `(polar polar)

In my opinion, leaving equ? up to the complex number package is a far better abstraction.

All these solutions are bad, because they require the modification of the original packages, and thus break additivity.

Liskov

My solution does not require to modifying the other packages. It does not work with mixed types, because I have used the generic operation 'sub'. That is fine, because the enunciate of the exercise do not specify that the 'equ?' operation should work with mixed types. But if you want, the code below can work with coercion procedures (like 'rational->complex'), or with 'apply-generic' of ex-2.82.

```
(define (equ? x y)
  ((get 'equ? 'generic) x y))

(define (install-generic-arithmetic-package)
  ; tolerance is  $\varepsilon * 4$ . If it was zero, the last test (that with the PI constant) would fail.
  (define tolerance
    (* ((lambda ()
      (define (try value)
        (let ((next-value (/ value 2)))
          (if (= (+ next-value 1) 1)
            value
            (try next-value))))
      (try 1.0)))
    4))
  ; imported procedures from rational package
  (define (numer x) ((get 'numer '(rational)) x))
  (define (denom x) ((get 'denom '(rational)) x))
  ; internal procedures
  (define (=zero? x)
    (and (<= (abs (real-part x)) tolerance) (<= (abs (imag-part x)) tolerance)))
  (define (equ? x y) (=zero? (sub x y)))
  ; interface to rest of the system
  (put 'real-part '(scheme-number) (lambda (x) x))
  (put 'imag-part '(scheme-number) (lambda (x) 0))
  (put 'real-part '(rational) (lambda (x) (/ (numer x) (denom x))))
  (put 'imag-part '(rational) (lambda (x) 0))

  (put 'equ? 'generic equ?)
  'done)

;; TESTING

(display (equ? (make-scheme-number (/ 1 3))
  (make-scheme-number 0.3333333333333333)))
(display (equ? (make-rational 2 11) (make-rational 2 10)))
(display (equ? (make-complex-from-mag-ang 5 (asin 0.8))
  (make-complex-from-real-imag 3 4)))
(display (equ? (make-rational 8.881784197001252e-16 1) (make-rational 0 1)))
(display (equ? (make-complex-from-mag-ang 5 2)
  (make-complex-from-mag-ang 5 (+ 2 (* 2 pi))))))

;; output will be #t#f#t#t#t
```

The above depends on numer and denom being exposed by the rational package, which they are not.

As for "breaking additivity", new types of numbers can be added at any time. The system as designed does not support adding arbitrary fundamental operations to an existing number type that depend on knowledge of the underlying implementation. One would have to either expose lower-level internals to the entire system or implement new operations by explicitly dealing with details of the implementation of a number type.

With the former, one is now mixing levels of abstraction. With the latter, one now has two implementations of a number type, which is brittle: What happens when the package's internal implementation changes? Leaving specific implementations to the specific packages makes the most sense to me.

Liskov

Thanks for your comment. I disagree that there are two implementations of numbers on my solution. The underlying representation of the number types remains the same.

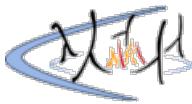
"What happens when the package's internal implementation changes?"

It doesn't matter as long as I'm using the "public interface" or the data abstraction - but 'numer' and 'denom' are not being exposed. Well, then there is no solution without modifying the original packages.

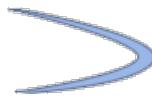
The problem is that I forgot something: **the generic arithmetic package is a superpackage of rational, complex and scheme number packages.** (see figure 2.23).

Therefore, when the exercise ask to install the predicate in the 'generic arithmetic package' it isn't imposing a restriction. I think the original packages should be modified, and the first solution, which puts an 'equ?' procedure in each subpackage, is correct.





# sicp-ex-2.80



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.79) | Index | Next exercise (2.81) >>

meteorgan

```
(define (=zero? x) (apply-generic '=zero? x))

;; add into scheme-number-package
(put '=zero? '(scheme-number) (lambda (x) (= x 0)))

;; add into rational-package
(put '=zero? '(rational)
     (lambda (x) (= (numer x) 0)))

;; add into complex-package
(put '=zero? '(complex)
     (lambda (x) (= (real-part x) (imag-part x) 0)))
```

Ada

A type-recursive version

```
; Version - 1
(define (=zero? data)
  (cond
    ((number? data) (= data 0))
    ((eq? 'rational (type-tag data))
     (=zero? (contents data)))
    ((eq? 'complex (type-tag data))
     (and (=zero? (real-part data))
          (=zero? (imag-part data)))))
    (else (error "xxx"))))

; Version -2
(define (=zero? data)
  (apply-generic '=zero? data))

(define (install-scheme-number-package)
  ...
  (put '=zero? '(scheme-number)
       (lambda (x) (= x 0))) ; scheme-number is the "base-type"

(define (install-rational-number-package)
  ...
  (put '=zero? '(rational)
       (lambda (x) (=zero? (numer x) 0))) ; the numer part might be another type of
                                         ; number, for example a rational number

(define (install-complex-number-package)
  ...
  (put '=zero? '(complex)
       (lambda (x)
         (and (=zero? (real-part x)) ; ibid
              (=zero? (imag-part x))))))) ; ibid
```

Reimu

It's worth noting that defining `=zero?` in terms of itself as the answer above does, allows for greater flexibility. For instance we can have a rational number whose `numer` is a complex number. However, I do have slight modification to the answer above:

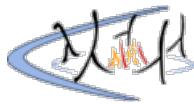
```
(put '=zero? '(rectangular) (lambda (x) (and (=zero? (real-part x))
                                             (=zero? (imag-part x)))))

(put '=zero? '(polar) (lambda (x) (=zero? (magnitude x)))))

(put '=zero? '(complex) =zero?) ; see ex 2.77. runs apply-generic twice.
```

This approach is more abstract, and is more efficient when dealing with polar numbers as there's no need to convert from one from to the other.

# sicp-ex-2.81



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.80) | Index | Next exercise (2.82) >>

meteorgan

**; ;a**  
apply-generic calls itself recursively on coerced types, so it goes into infinite recursion.

**; ;b**  
Louis's code can't work. apply-generic just works as it is.

**; ;c**  
`(define (apply-generic op . args)
 (define (no-method type-tags)
 (error "No method for these types"
 (list op type-tags)))

 (let ((type-tags (map type-tag args)))
 (let ((proc (get op type-tags)))
 (if proc
 (apply proc (map contents args))
 (if (= (length args) 2)
 (let ((type1 (car type-tags))
 (type2 (cadr type-tags))
 (a1 (car args))
 (a2 (cadr args)))
 (if (equal? type1 type2)
 (no-method type-tags)
 (let ((t1->t2 (get-coercion type1 type2))
 (t2->t1 (get-coercion type2 type1))
 (a1 (car args))
 (a2 (cadr args)))
 (cond (t1->t2
 (apply-generic op (t1->t2 a1) a2))
 (t2->t1
 (apply-generic op a1 (t2->t1 a2)))
 (else (no-method type-tags))))))
 (no-method type-tags))))`

atomik

meteorgan's answer to 87a is partially correct.

For scheme-numbers, it depends on which implementation of `(type-tag)` is being used. If (type-tag) is defined like so:

```
(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "Bad tagged datum")))
```

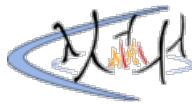
Then the program will exit with an error when apply-generic is called for the first time (it attempts to apply type-tag to untagged scheme primitives). However, if (type-tag) is able to handle scheme primitives like so:

```
(define (type-tag datum)
  (cond ((number? datum) 'scheme-number)
        ((pair? datum) (car datum))
        (else (error "Bad tagged datum"))))
```

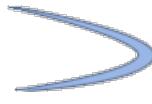
Then the program will loop recursively forever coercing scheme numbers into themselves and then calling apply-generic on the coerced data, as meteorgan says.

For 87b, I think that Louis is correct in the sense that "something had to be done about coercion with arguments of the same type" because even though (apply-generic) will exit with an error if no coercion is found, that doesn't stop our users from adding same-type coercion operators to our table (Louis already did it once). I do agree with meteorgan, that although something had to be done, the thing that Louis did is the wrong thing. I think apply-generic would be "more correct" if it was modified to exit with an error BEFORE attempting any same-type coercions. We could also add code to the `put` procedure which would make it exit with an error if a user attempted to `put`

in a same-type coercion, but that would be extremely difficult (maybe impossible) and also adversely affect the expressiveness of our language.



# sicp-ex-2.82



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.81) | Index | Next exercise (2.83) >>

jirf

If we assume the type hierarchy is a tower than types in the hierarchy will always converge onto the same type by promotion. To make sure that we promote all types to a minimum \*height in the tower and preserve the argument order (as it may be important to the procedure we are applying to) I chose a strategy inspired by pivot sort.

Like in pivot sort, break the argument list into 3 lists. Before current after. By promoting each element in before and after to current's type where possible, than recursing on \*right shifted before-current-after we maintain argument order. By the final recursion it is assured that all prior arguments to current are either the same type or a subtype which can be promoted to current (unless types are not in the same tower).

```
(define (apply-generic op . args)

(define (coerce object target-type)
  ;; if coerce object to target-type if possible, identity if not
  (let ((coercion (get-coercion (type-tag object) target-type)))
    (if coercion (coercion object) object)))

(define (iter before-reference reference after-reference)

(define (coerce-map objects)
  ;; convience mapping for coerce
  (map (lambda (object)
          (coerce object (type-tag (car reference))))
    objects))

;; reconstruct the full arg list for finding procedure and applying
(let ((args (append before-reference
                     reference
                     after-reference)))
  (let ((proc (get op (map type-tag args))))
    (cond
      ;; if procedure apply it
      ((not (null? proc))
       (proc (apply proc (map contents args))))
      ;; if we have no reference then no procedure for these types
      ((null? reference)
       (error "No method for these types"
             (list op (map type-tag args))))
      ;; if here we have reference
      ;; coerce types to reference type if possible and try again
      (else
       (let ((before-coerced (coerce-map before-reference))
             (after-coerced (coerce-map after-reference)))
         (cond
           ((null? after-reference)
            (iter (append before-coerced reference)
                  nil
                  nil))
           (else
            (iter (append before-coerced reference)
                  (list (car after-coerced))
                  (cdr after-coerced))))))))
      ;; start the process by setting reference to the first element
      (iter nil (list (car args)) (cdr args))))
```

2bdkiD

Only thing is this uses the primitive map, but otherwise I think it works correctly.

```
(define (any-false? items)
  (cond ((null? items) false)
        ((not (car items)) true)
        (else (any-false? (cdr items)))))
```

```

(define (coerce type-tags args)
  (define (iter tags)
    (if (null? tags)
        false
        (let ((type-to (car tags)))
          (let ((coercions
                  (map (lambda (type-from)
                          (if (eq? type-from type-to)
                              (lambda (x) ; identity "coercion" for same-types
                                (get-coercion type-from type-to))
                              type-tags)))
                (if (any-false? coercions)
                    (iter (cdr tags))
                    (map (lambda (coercion arg) (coercion arg))
                        coercions
                        args))))))
            (iter type-tags)))

(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (let ((coerced-args (coerce type-tags args)))
            (if coerced-args
                (let ((coerced-type-tags (map type-tag coerced-args)))
                  (let ((new-proc (get op coerced-type-tags)))
                    (apply new-proc (map contents coerced-args))))
                (error "No method for these types"
                      (list op type-tags)))))))

```

### Clean solution

woofy

```

(define (apply-generic op . args)

  (define (type-tags args)
    (map type-tag args))

  (define (try-coerce-to target)
    (map (lambda (x)
            (let ((coercion (get-coercion (type-tag x) (type-tag target))))
              (if coercion
                  (coercion x)
                  x)))
         args))

  (define (iterate next)
    (if (null? next)
        (error "No coercion strategy for these types" (list op (type-tags args)))
        (let ((coerced (try-coerce-to (car next))))
          (let ((proc (get op (type-tags coerced))))
            (if proc
                (apply proc (map contents coerced))
                (iterate (cdr next)))))))

  (let ((proc (get op (type-tags args))))
    (if proc
        (apply proc (map contents args))
        (iterate args)))

; Situation where this is not sufficiently general:
; types: A B C
; registered op: (op some-A some-B some-B)
; registered coercion: A->B C->B
; Situation: Evaluating (apply-generic op A B C) will only try (op A B C), (op B B B) and fail
; while we can just coerce C to B to evaluate (op A B B) instead

```



Exercise 2.82. Show how to generalize apply-generic to handle coercion in the general case of multiple arguments. One strategy is to attempt to coerce all the arguments to the type of the first argument, then to the type of the second argument, and so on. Give an example of a situation where this strategy (and likewise the two-argument version given above) is not sufficiently general. (Hint: Consider the case where there are some suitable mixed-type operations present in the table that will not be tried.)

Answer:

;iter returns the list of coerced argument or gives an error on failing to find a method.

```

(define (apply-generic op . args)
(define (iter type-tags args)
  (if (null? type-tags)
      (error "No method for these types-ITER")
      (let ((type1 (car type-tags)))
        (let ((filtered-args (true-map (lambda (x)
                                         {{scheme
                                             (let ((type2 (type-tag x)))
                                               (if (eq? type1 type2)
                                                   x
                                                   (let ((t2->t1 (get-coercion type2
type1)))
                                                     (if (null? t2->t1) #f (t2->t1 x)))))))
                                         (or filtered-args
                                             (iter (cdr type-tags) args))))))
          (let ((type-tags (map type-tag args)))
            (let ((proc (get op type-tags)))
              (if (not (null? proc))
                  (apply proc (map contents args))
                  (apply apply-generic (cons op (iter type-tags args)))))))
        ));; true-map function applies proc to each item on the sequence and returns false if any
of
;;;;those results was false otherwise returns the list of each results.
(define (true-map proc sequence)
  (define (true-map-iter proc sequence result)
    (if (null? sequence)
        (reverse result)
        (let ((item (proc (car sequence))))
          (if item
              (true-map-iter proc (cdr sequence) (cons item result))
              #f)))
  (true-map-iter proc sequence '())))
;; The following code may be used to try out the solution and see it working. Also you
will
;;;;need the scheme-number, rational and complex packages and associated generic
declarations
;;;;which are not given here.

(define *coercion-table* (make-equal-hash-table))

(define (put-coercion type1 type2 proc)
  (hash-table/put! *coercion-table* (list type1 type2) proc))

(define (get-coercion type1 type2)
  (hash-table/get *coercion-table* (list type1 type2) '()))

(define (install-coercion-package)
(define (scheme-number->complex n)
  (make-complex-from-real-imag (contents n) 0))
(define (scheme-number->rational n)
  (make-rational (contents n) 1))
(put-coercion 'scheme-number 'rational scheme-number->rational)
(put-coercion 'scheme-number 'complex scheme-number->complex)
'done)

(install-coercion-package)

;;The following are some example evaluations

;;RESULT
;; 1 ]=> (add (make-scheme-number 1) (make-scheme-number 4))

;; ;Value: 5

;; 1 ]=> (add (make-complex-from-real-imag 1 1) (make-complex-from-real-imag 3 2))

;; ;Value 14: (complex rectangular 4 . 3)

;; 1 ]=> (add (make-scheme-number 1) (make-complex-from-real-imag 1 1))

;; ;Value 15: (complex rectangular 2 . 1)

;; 1 ]=> (add (make-scheme-number 2) (make-rational 3 4))

;; ;Value 16: (rational 11 . 4)

;; 1 ]=>

```

meteorgan

```
(define (apply-generic op . args)
  ;; if all types can coerced into target-type
  (define (can-coerced-into? types target-type)
    (andmap
      (lambda (type)
        (or (equal? type target-type)
            (get-coercion type target-type))))
      types))
  ;; find one type that all other types can coerced into
  (define (find-coerced-type types)
    (ormap
      (lambda (target-type)
        (if (can-coerced-into? types target-type)
            target-type
            #f)))
      types))
  ;; coerced args into target-type
  (define (coerced-all target-type)
    (map
      (lambda (arg)
        (let ((arg-type (type-tag arg)))
          (if (equal? arg-type target-type)
              arg
              ((get-coercion arg-type target-type) arg))))
      args))
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (let ((target-type (find-coerced-type type-tags)))
            (if target-type
                (apply apply-generic
                      (append (list op) (coerced-all target-type)))
                (error "no method for these types" (list op type-tags))))))))
```

Ivan

In general it coerces argument list to the types in order from first to last and then tries to find a procedure to apply on these arguments. If coercion for pair of types is not found I just put non-coerced element in the coerced list which allows for defining mixed-type procedures. This way I avoid some logic and make things work by convention.

What needs to be done in the rest of the system is to define generic methods to be applicable on arbitrary number of arguments, since in this point our arithmetic operations work only on 1 or 2 args.

```
(define (apply-generic op . args)
  ; coercing list to a type
  (define (coerce-list-to-type lst type)
    (if (null? lst)
        '()
        (let ((t1->t2 (get-coercion (type-tag (car lst)) type)))
          (if t1->t2
              (cons (t1->t2 (car lst)) (coerce-list-to-type (cdr lst) type))
              (cons (car lst) (coerce-list-to-type (cdr lst) type)))))

  ; applying to a list of multiple arguments
  (define (apply-coerced lst)
    (if (null? lst)
        (error "No method for given arguments")
        (let ((coerced-list (coerce-list-to-type args (type-tag (car lst)))))
          (let ((proc (get op (map type-tag coerced-list)))
                (if proc
                    (apply proc (map contents coerced-list))
                    (apply-coerced (cdr lst))))))

  ; logic to prevent always coercing if there is already direct input entry
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (apply-coerced args))))
```

Wing

the above coerce-list-to-type can be replaced with

```
(map (lambda (x)
  (let ((proc (get-coercion (type-tag x) type)))
    (if proc (proc x) x)))
```

```
lst)
```

Xavier

```
(define (identity x) x)
(define (apply-generic . args)

  (define (inner op . args)
    (let ((type-tags (map type-tag args)))
      (let ((proc (get op type-tags)))
        (if proc
            (apply proc (map contents args))
            (if (coercion-possible? type-tags)
                (let ((coercions (get-coercions type-tags)))
                  (if coercions
                      (apply inner (cons op (map (lambda (coercion value)
                                         (coercion value))
                                         coercions
                                         args)))
                      (error "No method for these types" (list op types))))
                (error "No method for these types" (list op types)))))

  ;; Types can be coerced as long as there are types to coerce and there is at least one
  type
  ;; to coerce
  (define (coercion-possible? types)
    (not (or
          (null? types)
          (every (lambda (type) (equal? type (car types))) types)))

  (define (get-coercions types)
    ;; Retrieves the coercion functions for each of the provided types, using (car base-
    types)
    ;; as the base type.
    ;; If a coercion function for all requested types to (car base-types) is not found,
    then
    ;; try with the next base-type until the list is exhausted.
    ;
    ;; The first parameter is an ordered list of coercion functions.
    ;; The second parameter is a list of the remaining types in need of a coercion
    function
    ;; The third parameter is the list of base types we can use to attempt coercion
    (define (iter coercions to-coerce base-types)
      (cond ((null? to-coerce) coercions)
            ((null? base-types) (error "No method for these types" types))
            (else
              (let ((type (car to-coerce))
                    (base-type (car base-types)))
                (cond ((equal? type base-type)
                      (iter (append coercions (list identity))
                            (cdr to-coerce)
                            base-types))
                      ((get-coercion type base-type)
                        (iter (append coercions (list (get-coercion type base-type)))
                              (cdr to-coerce)
                              base-types))
                      (else (iter '() types (cdr base-types)))))))
              (iter '() types types))

  (apply inner args))
```

poly

A solution more readable and efficient than meteorgan's.

```
(define (filter proc seq)
  (cond ((null? seq) nil)
        ((proc (car seq))
         (cons (car seq) (filter proc (cdr seq))))
        (else
         (filter proc (cdr seq)))))

(define (apply-generic op . args)
  ; find the type that all args can be coerced into, through
  ; filtering the given types.
  (define (find-generic-type arg-types type-tags)
    (cond ((null? type-tags) #f)
          ((null? arg-types) (car type-tags))
          ; use car instead of returning the whole list because
          (else (find-generic-type (cdr arg-types) (cdr type-tags))))))
```

```

; normally, there will be only one existed type.
(else
  (find-generic-type (cdr args)
    (find-coercion-types (car args)
      type-tags)))))

; find the types that the arg can be coerced into among given
; types; this will return a list due to the filter procedure
(define (find-coercion-types arg-type type-tags)
  (filter (lambda (t2)
    (or (equal? arg-type t2)
        (get-coercion arg-type t2)))
  type-tags))

; coerce all the args into target-type
(define (coerce-all target-type)
  (map (lambda (arg)
    (let ((arg-type (type-tag arg)))
      (if (equal? arg-type target-type)
          arg
          (get-coercion arg-type target-type) arg))))
  args))

(let ((type-tags (map type-tag args)))
  (let ((proc (get op type-tags)))
    (if proc
        (apply proc (map contents args))
        (let ((target-type (find-generic-type args)))
          (if target-type
              (apply apply-generic
                (cons op (coerced-all target-type)))
              (error "no method for these types"
                (list op type-tags)))))))

```

joe w

One of my problems with sicp is figuring out what the authors expect you to do. It seems like a lot of solutions on this page and and elsewhere on the web only work when you update your internal procedures to have a list of more arguments, for example (add '(complex complex complex)) to support 3 arguments. But what if I want to type 4 args or 5 or more? I have to add more entries?

My solution works just by altering apply-generic and using nothing more than the two argument internal functions the authors original present such as (add '(complex complex)).

```

(define (apply-gen op . args)

  (define (coerce-all-args op args)

    (define (coerce-to-single-type arg all-args result)
      (let ((type (type-tag arg))
            (type-tags (map type-tag all-args)))
        (if (null? all-args) (reverse result)
            (if (eq? type (car type-tags))
                (coerce-to-single-type arg (cdr all-args) (cons (car all-args) result))
                (let ((proc (get-coercion (car type-tags) type)))
                  (if proc
                      (coerce-to-single-type arg (cdr all-args) (cons (proc (car all-args))
result))
                      #f)))))

    (define (apply-to-all args op)
      (displayln args)
      (define (iter args result)
        (if (null? args) result
            (let ((value (contents (car args))))
              (iter (cdr args) (if (null? result)
                value
                (op result value))))))

      (iter args '())))

;wrapper function to iterate
(define (check-all remaining-args)
  (if (null? remaining-args)
      (error "no suitable coercion operations found")
      (let ((coerced-args (coerce-to-single-type (car remaining-args) args '())))
        (arg-type (type-tag (car remaining-args))))
      (if coerced-args
          (apply-to-all coerced-args (get op (list arg-type arg-type)))
          (check-all (cdr remaining-args)))))

  (check-all args))

  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (coerce-all-args op args)))))


```

It's not pretty and it could be more efficient since it does the coercion before it even knows if all the items in a list can be coerced, but at least I don't have to go back and add an item to the internal procedures.

This was written using DrRacket 6.12. lang #racket

Sphinxsky

Although my solution is inefficient, it should be clearer.

```
; Import an accumulator
(load "accumulate.scm")

(define (apply-generic op . args)

  ; Throws an exception to a procedure call
  (define (no-method-error tags)
    (error
      "No method for these types"
      (list op tags)))

  ; Mandatory conversion process table for tabulating parameters
  ; Returns a two-dimensional process table
  ; Item quantity: (square (length args))
  (define (coercion-proc-table args)
    (map
      (lambda (x)
        (let ((type-x (type-tag x)))
          (map
            (lambda (y)
              (let ((type-y (type-tag y)))
                (if (eq? type-x type-y)
                  (lambda (this) this)
                  (get-coercion type-y type-x))))
            args)))
      args))

  ; Searching for Conversion Processes Sequences
  ; If not, return null
  (define (find-proc-list table)
    (filter
      (lambda (proc-seq)
        (accumulate and #t proc-seq))
      table))

  ; Conversion parameter list
  (define (coercion-transform proc-seq args)
    (map
      (lambda (f x) (f x))
      proc-seq
      args))

  ; Main logic
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
        (apply proc (map contents args))
        (let ((proc-list (find-proc-list (coercion-proc-table args))))
          (if (null? proc-list)
            (no-method-error type-tags)
            (let ((new-args (coercion-transform (car proc-list) args)))
              (let ((new-type-tags (map type-tag new-args)))
                (if (equal? new-type-tags type-tags)
                  (no-method-error type-tags)
                  (apply apply-generic (cons op new-args)))))))))))
```

revc

Version with counter

```
#lang sicp
(define (apply-generic op . args)

  (define (no-method type-tags)
    (error
      "No method for these types"
      (list op type-tags)))
```

```

(define (apply-generic-count op counter . args)
  (let ((type-tags (map type-tag args)))
    (if (= counter 0)
        (no-method type-tags)
        (let ((proc (get op type-tags)))
          (if proc
              (apply proc (map contents args))
              (let ((coercion-list (get-coercion-list (car type-tags) (cdr type-
tags))))
                (if (valid? coercion-list)
                    (apply-generic-count op counter
                        (car args)
                        (apply-coercion-list (cdr args)))
                    (apply-generic-count op (dec counter)
                        (cdr args)
                        (car args)))))))))

(apply-generic-count op (length args) args)

(define (get-coercion-list type type-list)
  (map (lambda (x) (get-coercion x type)) type-list))

(define (valid? coercion-list)
  (accumulate and true coercion-list))

(define (apply-coercion-list coercion-list type-list)
  (map (lambda (f x) f x) coercion-list type-list))

```

Hatsune Miku

I think my solution is rather elegant, and easy to read. My approach is to generate a list of coercing procedures and then use map on the list of argument.

```

;; any null in the list?
(define (has-null? seq)
  (memq '() seq))

;; Generate a list of coercion procedures, and the apply each one to the
;; corresponding arguement.
(define (coerce args-list)
  (define (coerce-helper remaining)
    (cond ((null? remaining) (error "Not possible to coerce"))
          (else
            (let ((master-arg (car remaining)))
              (let ((master-type (type-tag master-arg)))
                (let ((procedure-list
                      (map (lambda (somearg)
                            (let ((sometype (type-tag somearg)))
                              (if (eq? sometype master-type) (lambda (x) x) ; if they
                                have the same type, then subsitute with the identity
                                (get-coercion sometype master-type)))))))
                  (args-list)))
                  (if (has-null? procedure-list) ; is the coercion procedures list
                      valid?
                      (coerce-helper (cdr remaining)) ; try the next master-type
                      (map apply procedure-list args-list)))))))) ; apply the nth
coercion to the nth argument
(coerce-helper args-list))

;; This startegy won't work unless we explecitly define coercions of the form
;; t1->t3, we can't for instance use t1->t2 and then t2->t3 without defining
;; them first

;; does the list have any #f?
(define (has-false? seq)
  (memq #f seq))

;; are all elements of the list the same?
(define (all-same? seq)
  (cond ((null? seq) true)
        (else
          (let ((first (car seq)))
            (has-false?
              (map (lambda (somearg)
                    (eq? somearg first)))))))

;; if `proc` does not exist, and the arguments are NOT of the same type, then
;; try apply-generic with the coerced arguements
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (cond (proc (apply proc (map contents args)))
            ((all-same? type-tags)

```

```

(error "No method for these types" (list op type-tags)))
(else
  (apply-generic op (coerce args))))))

```

Paj

My approach is to coerce the list args into a coerced list of args of equal type. The coercion of an individual arg is done in the same way as the two arg version given in the book.

```

(define (coerce-lst-from-to uncoerced coerced)
  (if (null? uncoerced)
    coerced
    (let*
      ((coerced-item      (car coerced))
       (uncoerced-item   (car uncoerced))
       (l-uncoerced-item (list uncoerced-item))

       (type-c            (type-tag coerced-item))
       (type-u            (type-tag uncoerced-item))
       (same-type?        (eq? type-c type-u))

       (tu->tc           (if (not same-type?)
                               (get-coercion type-u type-c)
                               false))
       (tc->tu           (if (not tu->tc)
                               (get-coercion type-c type-u)
                               false))

       (new-coerced-lst   (cond (same-type?
                                 (append coerced l-uncoerced-item))

                                 (tu->tc
                                   (append coerced (list (tu->tc uncoerced-item)))))

                                 (tc->tu
                                   (append (map tc->tu coerced) l-uncoerced-item))

                                 (else false)))

       (new-uncoerced-lst (cdr uncoerced)))

      (if new-coerced-lst
        (coerce-lst-from-to new-uncoerced-lst new-coerced-lst)
        false)))))

(define (coerce-lst lst)
  (if (null? lst)
    lst
    (coerce-lst-from-to (cdr lst) (list (car lst)))))

(define (eq-types-from-to type type-lst)
  (if (null? type-lst)
    true
    (if (eq? type (car type-lst))
      (eq-types-from-to type (cdr type-lst))
      false)))

(define (eq-types? type-tags)
  (if (null? type-tags)
    false
    (if (= (length type-tags) 1)
      true
      (eq-types-from-to (car type-tags) (cdr type-tags)))))

;; Apply-generic with coercion
(define (apply-generic op . args)
  (let* ((type-tags (map type-tag args))
         (proc (get op type-tags)))
    (if proc
      (apply proc (map contents args))
      (if (and (> (length args) 1) (not (eq-types? type-tags)))
        (let ((coerced-args (coerce-list args)))
          (if coerced-args
            (apply apply-generic (cons op coerced-args))
            (error
              "No method for types [2]"
              (list op type-tags))))
        (error
          "No method for types [1]"
          (list op type-tags))))))

```



# sicp-ex-2.83

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.82) | Index | Next exercise (2.84) >>

meteorgan

```
(define (raise x) (apply-generic 'raise x))

;; add into integer package
(put 'raise '(integer)
     (lambda (x) (make-rational x 1)))

;; add into rational package
(put 'raise '(rational)
     (lambda (x) (make-real (* 1.0 (/ (numer x) (denom x))))))

;; add into real package
(put 'raise '(real)
     (lambda (x) (make-complex-from-real-imag x 0)))
```

leafac

One other way to do the same is use the coercion table:

```
(define (integer->rational integer)
  (make-rational integer 1))
(define (rational->real rational)
  (define (integer->floating-point integer)
    (* integer 1.0))
  (make-real (/ (integer->floating-point (numer rational))
                (denom rational))))
(define (real->complex real)
  (make-complex-from-real-imag real 0))
>>>
(put-coercion 'integer 'rational integer->rational)
(put-coercion 'rational 'real rational->real)
(put-coercion 'real 'complex real->complex)

(define (raise number)
  (define tower '(integer rational real complex))
  (define (try tower)
    (if (< (length tower) 2)
        (error "Couldn't raise type" number)
        (let ((current-type (car tower))
              (next-types (cdr tower))
              (next-type (car next-types)))
          (if (eq? (type-tag number) current-type)
              ((get-coercion current-type next-type) number)
              (try next-types)))))

  (try tower))
```

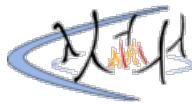
master

I have to say, these exercises are rather difficult because it's not clear to what extent we should be creating a working system. I see that a lot of people here have been implementing everything so that all the code runs, but the feeling I get from the book is that we only need to write procedures which work in theory and y'know leave everything else up to ol' George. So anyway, I have sort of been skimming these exercises and looking at them as puzzles rather than an actual program. Anyway, this procedure doesn't really do anything because I don't have a table and can't look up any procedures, so I decided to just not bother, it is trivial to get the actual procedures, this is just the raising mechanism.

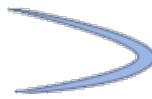
```
(define tower '(integer rational real complex))

(define (raise type)
  (let ((position (memq type tower)))
    (if (eq? position #f)
        (error "No such datatype: " type)
```

```
(let ((this-type (car position)))
  (let ((rest (cdr position)))
    (if (null? rest)
        this-type
        (let ((next-type (cadr position))
              next-type))))))
```



# sicp-ex-2.84



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.83) | Index | Next exercise (2.85) >>

meteorgan

```
; assuming that the types are arranged in a simple tower shape.  
(define (apply-generic op . args)  
  ; raise s into t, if success, return s; else return #f  
  (define (raise-into s t)  
    (let ((s-type (type-tag s))  
          (t-type (type-tag t)))  
      (cond ((equal? s-type t-type) s)  
            ((get 'raise (list s-type))  
             (raise-into ((get 'raise (list s-type)) (contents s)) t))  
            (else #f)))  
  
(let ((type-tags (map type-tag args)))  
  (let ((proc (get op type-tags)))  
    (if proc  
        (apply proc (map contents args))  
        (if (= (length args) 2)  
            (let ((a1 (car args))  
                  (a2 (cadr args)))  
              (cond  
                  ((raise-into a1 a2)  
                   (apply-generic op (raise-into a1 a2) a2))  
                  ((raise-into a2 a1)  
                   (apply-generic op a1 (raise-into a2 a1)))  
                  (else (error "No method for these types"  
                               (list op type-tags))))  
            (error "No method for these types"  
                  (list op type-tags)))))))
```

sam

There is a bug in meteorgan's solution. The bug arises for same reason as exercise 2.81. That is, for an 'op' not defined for identical types a1 & a2, there is infinite recursion.

gws

```
; This solution is for an apply-generic that works with arbitrary arguments  
; by raising all to the highest type.  
; New types can be added to the tower easily. Only the procedure "level" has  
; to be modified.  
  
(define (level type)  
  (cond ((eq? type 'integer) 0)  
        ((eq? type 'rational) 1)  
        ((eq? type 'real) 2)  
        ((eq? type 'complex) 3)  
        (else (error "Invalid type: LEVEL" type))))  
  
(define (apply-generic op . args)  
  (let ((type-tags (map type-tag args)))  
    (define (no-method)  
      (error "No method for these types" (list op type-tags)))  
    (let ((proc (get op type-tags)))  
      (if proc  
          (apply proc (map contents args))  
          (if (not (null? (cdr args))) ; length of args > 1  
              (let ((raised-args (raise-to-common args)))  
                (if raised-args  
                    (let ((proc (get op (map type-tag raised-args))))  
                      (if proc  
                          (apply proc (map contents raised-args))  
                          (no-method)))  
                    (no-method))))  
          (no-method))))
```

```

(define (raise-to-common args)
  (let ((raised-args
         (map (lambda (x) (raise-to-type (highest-type args) x))
              args)))
    (if (all-true? raised-args)
        raised-args
        false)))

(define (all-true? lst)
  (cond ((null? lst) true)
        ((car lst) (all-true? (cdr lst)))
        (else false)))

(define (raise-to-type type item)
  (let ((item-type (type-tag item)))
    (if (eq? item-type type)
        item
        (let ((raise-fn (get 'raise item-type)))
          (if raise-fn
              (raise-to-type type (raise-fn item))
              false)))))

(define (highest-type args)
  (if (null? (cdr args))
      (type-tag (car args))
      (let ((t1 (type-tag (car args)))
            (t2 (highest-type (cdr args))))
        (let ((l1 (level t1)) (l2 (level t2)))
          (if (> l1 l2) t1 t2)))))


```

### Rptx

```

; This is very similar to gws' answer, but, I have added the levels to the
; table, so new levels can be added to the table in the new types' package
; using "put"

(define (apply-generic op . args)
(define (higher-type types)
  (define (iter x types)
    (cond ((null? types) x)
          ((> (level x) (level (car types)))
           (iter x (cdr types)))
          (else
            (iter (car types) (cdr types)))))
  (if (null? (cdr types)) (car types) (iter (car types) (cdr types))))
(define (raise-args args level-high-type)
  (define (iter-raise arg level-high-type)
    (if (< (level (type-tag arg)) level-high-type)
        (iter-raise (raise arg) level-high-type)
        arg))
  (map (lambda (arg) (iter-raise arg level-high-type)) args))
(define (not-all-same-type? lst)
  (define (loop lst)
    (cond ((null? lst) #f)
          ((eq? #f (car lst)) #t)
          (else (loop (cdr lst))))))
  (loop (map (lambda (x) (eq? (car lst) x))
             (cdr lst)))))

(let ((type-tags (map type-tag args)))
  (let ((proc (get op type-tags)))
    (if proc
        (apply proc (map contents args))
        (if (not-all-same-type? type-tags) ; only raise if the args are not
            ; all of the same type, if they are, then there is not "op" for them.
            (let ((high-type (higher-type type-tags)))
              (let ((raised-args (raise-args args (level high-type))))
                (apply apply-generic (cons op raised-args))))
            (error
              "No Method for these types -- APPLY-GENERIC"
              (list op type-tags)))))

; The level procedure.

(define (level type)
  (get 'level type))

; The package of the levels. Each of these statements, can be placed in the
; package of the number type it belongs to. I have done it here for the
; exercise. Also I have a polar and rectangular level just in case level gets
; called with a complex number without the 'complex tag.

```

```

(define (install-level-package)
  (put 'level 'scheme-number 1)
  (put 'level 'rational 2)
  (put 'level 'real 3)
  (put 'level 'complex 4)
  (put 'level 'rectangular 4)
  (put 'level 'polar 4)
  'done)

(install-level-package)

```

bma

*;Hello here I'd like to introduce my own view on this task. No additional level tags, everything done with respect to the additive style of system construction and inside the operation-and-type-table and coercion-table boundaries. Also pay attention to the type-tag selectors/constructors and keep in mind that type constructors are globally announced, also the numer/denom from now are generic operations since we would need them for reals also to push down real to rational.*

```

;operation-and-type table
(define mt (make-hash))
;put procedure
(define (put op type proc) (hash-set! mt (cons op type) proc))
;get procedure
(define (get op type)
  (let ((pick-item (hash-ref mt (cons op type) '())))
    (if (null? pick-item)
        #f
        pick-item)))

;type-tag operations
(define (contents datum)
  (cond ((number? datum) datum)
        ((pair? datum) (cdr datum))
        (else (error "Bad data"))))
(define (type-tag datum)
  (cond ((number? datum) (if (exact? datum) 'integer 'real))
        ((pair? datum) (car datum))
        (else (error "Bad data"))))
(define (attach-tag type-tag contents)
  (if (number? contents)
      contents
      (cons type-tag contents)))

;global numer denom selectors
(define (numer x)
  (apply-generic 'numer x))
(define (denom x)
  (apply-generic 'denom x))

;coercion table
(define ct (make-hash))

;put-coercion procedure
(define (put-coercion type1 type2 proc) (hash-set! ct (cons type1 type2) proc))

;get-coercion procedure
(define (get-coercion type1 type2)
  (let ((pick-item (hash-ref ct (cons type1 type2) '())))
    (if (null? pick-item)
        #f
        pick-item)))

;basic coercions (global procedures). trick here is that rationals and reals have the
;"hidden" tag plus numer/denom are now generic and they will cut our rational in pieces and
;strip off the tag, so it's no more necessary to strip off the tag at the 'raising' step.
(define (integer->rational x)
  (make-rational x 1))
(define (rational->real x)
  (make-real (/ (numer x) (denom x))))
(define (real->complex x)
  (make-complex-from-real-imag x 0))

(define (install-coercions)
  (put-coercion 'integer 'rational (lambda (x) (integer->rational x)))
  (put-coercion 'rational 'real (lambda (x) (rational->real x)))
  (put-coercion 'real 'complex (lambda (x) (real->complex x))))
(install-coercions)

;raising (basically it will return its input value in case the type is 'complex, at this

```

```

point of the book, i found it more convenient to do it this way rather than just return
false, reason was to simplify the level-test)
(define (raise obj)
  (let ((type-tag (type-tag obj)))
    (let ((supertype (get 'raise type-tag)))
      (if supertype
          ((get-coercion type-tag supertype) obj)
          obj)))

;raising to the prespecified type (the issue is known that it will go into an infinite
loop in case the type is lower than argument's. but still it works fine as a part of apply-
generic procedure)
(define (raise-to arg type)
  (let ((this-tag (type-tag arg)))
    (if (eq? this-tag type)
        arg
        (raise-to (raise arg) type)))))

;testing 2 arguments which has the highest type
(define (pick-higher arg1 arg2)
  (define (find-iter arg1 arg2 result)
    (let ((tag1 (type-tag arg1))
          (tag2 (type-tag arg2))
          (move-a1 (raise arg1)))
      (let ((next-tag1 (type-tag move-a1)))
        (cond ((eq? tag1 tag2) arg2)
              ((eq? tag1 next-tag1) result)
              (else (find-iter move-a1 arg2 result)))))))
  (find-iter arg1 arg2 arg1))

;picking the highest type argument from the list
(define (find-highest args)
  (if (null? (cdr args))
      (car args)
      (let ((this (car args))
            (next (cadr args))
            (rest (cddr args)))
        (let ((t1 (type-tag this))
              (t2 (type-tag next)))
          (if (eq? t1 t2)
              (find-highest (cdr args))
              (find-highest (cons (pick-higher this next)
                                  rest)))))))

;raising the entire argument list to the type
(define (raise-all-to-highest args type)
  (if (null? args)
      null
      (let ((al (car args))
            (rest (cdr args)))
        (cons (raise-to al type)
              (raise-all-to-highest rest type)))))

;special procedure to partition our argument list in pieces of two, since our packages
arithmetic procedures were specified for maximum of 2 arguments, we'll do it this way:
(define (partition-and-apply op args)
  (if (null? (cdr args))
      (car args)
      (let ((a1 (car args))
            (a2 (cadr args))
            (rest-args (cddr args)))
        (partition-and-apply op (cons (apply-generic op a1 a2) rest-args)))))

;here we just construct the whole procedure. and it works properly for 1,2 or more
arguments and thanks god returns "Bad data" error in case the argument list is null. i
confess it could be written in a more 'generic' manner but if you find anything useful for
yourself out here i would be glad.
(define (apply-generic1 op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (if (> (length args) 1)
              (let ((t1 (car type-tags))
                    (t2 (cadr type-tags))
                    (rest-args (cddr args)))
                (if (and (null? rest-args) (eq? t1 t2))
                    (error "No procedure specified for these types" op)
                    (let ((highest-type (type-tag (find-highest args))))
                      (let ((raised-args (raise-all-to-highest args highest-type)))
                        (partition-and-apply op raised-args))))))
          (error "No procedure specified for this type" op)))))


```

```

;; Making a Hierarchical Manager
(define (make-rank-manager)
  ;; Type of hierarchical tower
  ;; The higher the level, the larger the coding.
  (define type-tower (list))

  ;; Create hierarchical objects
  (define (make-floor type hierarchy)
    (cons type hierarchy))

  ;; Get the hierarchy of hierarchical objects
  (define (get-hierarchy f) (cdr f))

  ;; Get the type of hierarchical object
  (define (get-type f) (car f))

  ;; Find the hierarchy of the type
  (define (find-hierarchy type)
    (let ((this (filter
                  (lambda (f)
                    (eq? type (get-type f)))
                  type-tower)))
      (if (null? this)
          (error "Non-existent type--" type)
          (get-hierarchy (car this)))))

  ;; Find the hierarchy of the type
  (define (find-type hierarchy)
    (let ((this (filter
                  (lambda (f)
                    (eq? hierarchy (get-hierarchy f)))
                  type-tower)))
      (if (null? this)
          (error "Non-existent hierarchy --" hierarchy)
          (get-type (car this)))))

  ;; Add level
  (define (append-floor! type hierarchy)
    (let ((same (filter
                  (lambda (t) (eq? type t))
                  (map get-type type-tower))))
      (if (null? same)
          (set!
            type-tower
            (cons (make-floor type hierarchy) type-tower))
          (error "existent type--" type)))))

  ;; selecting operation
  (define (operation op)
    (cond ((eq? op 'find-hierarchy) find-hierarchy)
          ((eq? op 'find-type) find-type)
          ((eq? op 'append-floor!) append-floor!)
          (else (error "Non-existent operation--" op)))))

  operation)

;; Operation interface
(define rank-manager (make-rank-manager))
(define get-rank (rank-manager 'find-hierarchy))
(define get-type (rank-manager 'find-type))
(define put-rank (rank-manager 'append-floor!))

;; Create a Set Type Tower
;; Layer hierarchy is controlled by a basic numerical value
;; The larger the value, the higher the level.
(define (make-number-tower)
  (put-rank 'scheme-number 1)
  (put-rank 'rational 3)
  (put-rank 'real 5)
  (put-rank 'complex 7)
  'done)

;; Promote data to a given type hierarchy
(define (raise-it data hierarchy)
  (let ((rank (get-rank (type-tag data))))
    (if (= hierarchy rank)
        data
        (raise-it (raise data) hierarchy)))))

(define (apply-generic op . args)

```

```

(define (no-method-error tags)
  (error
    "No method for these types"
    (list op tags)))

(let ((type-tags (map type-tag args)))
  (let ((proc (get op type-tags)))
    (if proc
        (apply proc (map contents args))
        (let ((type-top
              (apply max (map get-rank type-tags))))
          (let ((new-args
                 (map (lambda (x) (raise-it x type-top)) args)))
            (let ((new-type-tags (map type-tag new-args)))
              (if (equal? new-type-tags type-tags)
                  (no-method-error type-tags)
                  (apply apply-generic (cons op new-args)))))))))))

```

Spraynard

This solution should be able to supply an apply-generic procedure that takes in multiple arguments and raises each of the arguments to a type in which they can all be raised to. Performs the same operations as is seen, most likely, on a solution for Exercise 2.82.

Additivity of types seems to still work, but the problem with this implementation is that it cannot support additions of differently named types that are on the same "level" as other types..

```

;; Type heights are based on the index of the type in this list. '(complex)es
;; are "higher" in the tree than '(integers) because they are index 3 and
;; '(integers) are index 0.

(define type-height-relationships '(integer rational real complex))

;; Get the numerical index of item in list.
(define (index item itemlist)
  (define (index-loop count items)
    (cond ((null? items) #f)
          ((and (list? (car items))
                (index item (car items)) count)
           ((eq? item (car items)) count)
           (else (index-loop (+ count 1) (cdr items))))))
  (index-loop 0 itemlist))

;; Compares two type heights in the tower
;; Returns -1 if a < b
;; Returns 0 if a == b
;; Returns 1 if a > b
(define (type-compare a b)
  (let ((height-a (index (type-tag a) (type-height-relationships)))
        (height-b (index (type-tag b) (type-height-relationships))))
    (if (or (not height-a) (not height-b))
        (error "ERROR: Unknown type in type-height-relationships
given -- TYPE-COMPARE\nCheck your \"type-height-relationships\"")
        (cond ((< height-a height-b) -1)
              ((= height-a height-b) 0)
              ((> height-a height-b) 1)))))

;; Single parameter function that returns a single parameter lambda function.
;; We raise the secondary parameter (typed-to-coerce) up to the
;; level of the primary parameter (typed) through iterative recursion.
;;
;; If we cannot raise the secondary param then we return it and try to force
;; our apply-generic procedure to give us an operation that works.
(define (coerce-from-typed typed)
  (define (coerce-typed typed-to-coerce)
    (let ((typed-comparison (type-compare (type-tag typed)
                                         (type-tag typed-to-coerce))))
      (cond
        ;; Just return our typed-to-coerce and try to force a "good"
        ;; operation find. Ideally we'd like to "lower" the
        ;; "typed-to-coerce" here but we will assume that is not
        ;; available.
        ((< typed-comparison 0) typed-to-coerce)
        ;; Preferable exit method of our function.
        ((= typed-comparison 0) typed-to-coerce)
        ;; Recursing point of the function.
        ;; Raise our typed-to-coerce one more level and
        ;; do another test
        ((> typed-comparison 0) (coerce-typed (raise typed-to-coerce)))))))

```

```

(lambda (type) (coerce-typed type)))

;; Customized apply-generic procedure.
(define (apply-generic op . args)
  (define (apply-generic-loop op refs args)
    (if (null? refs)
        (error "No operation available for these types: "
              (map type-tag args)
              "\n"
              "Operation: " op)
        (let ((convert-to (coerce-from-typed (car refs))))
          (let ((converted (map convert-to args)))
            (let ((proc (get op (map type-tag converted))))
              (if proc
                  (apply proc (map contents converted))
                  (apply-generic-loop op (cdr refs) args)))))))
  (apply-generic-loop op args args))

```

Marisa

```

;; any null in the list?
(define (has-null? seq)
  (memq '() seq))

;; does the list have any #f?
(define (has-false? seq)
  (memq #f seq))

;; are all elements of the list the same?
(define (all-same? seq)
  (cond ((null? seq) true)
        (else
          (let ((first (car seq)))
            (has-false?
              (map (lambda (somearg)
                     (eq? somearg first)))))))

;; tower procedures
(define (superiors x)
  (cdr (memq x (find-tower x)))) ; I'm assuming find-tower exists

(define (above? a-type b-type)
  (memq b-type (find-tower a-type)))

(define (next-type type)
  (car (superiors type)))

;; be aware that raise takes a datum as input, not a tag
(define (raise datum)
  (let ((tag (type-tag datum)))
    ((get-coercion tag (next-type tag)) datum)))

;; successively rises one argument until it reaches the desired type
(define (successive-rise datum newtype)
  (if (eq? (type-tag datum) newtype) datum
      (successive-rise (rise datum) newtype)))

; find the highest tag in a list
(define (highest-type tag-list)
  (define (helper remaining)
    (cond ((null? remaining) (error "Bad tags" tag-list))
          (else
            (let ((master-tag (car remaining)))
              (all-true? (lambda (x) (not (has-false? x)))))
              (if (all-true? (map (lambda (sometag) (above? master-tag sometag)
                                    tag-list)))
                  master-tag
                  (helper (cdr remaining))))))) ; try again
  (helper tag-list))

;; If types are not the same, and the procedure does not exist, then run
;; apply-generic on the coerced version of the args, which are obtained by
;; mapping successive-raise on each arguments till they reach the highest-type
;; in tag-list
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (cond (proc (apply proc (map contents args)))
            ((all-same? type-tags)
             (error "No method for these types" (list op type-tags)))
            (else
              (let ((master-type (highest-type type-tags)))
                (let ((coerced-args (map (lambda (somearg)

```

```

        (successive-rise somearg master-type))
        args-list)))
(apply-generic op coerced-args)))))))

;;

;; Bonus! I have noticed that both the `coerce` procedure of 2.82 and the
;; `highest-type` procedure of 2.84 look simillar, which means that they could
;; both be defined in terms of a common abstraction barrier.

;; By examing their from we can see that they accpet a list as an arguement, and
;; then run map with a procedure that involves the car on the whole list.
;; Finally it runs a predicate, on the resulting map list, and if it satisfies
;; it, we return the car, if not, we try again with the cdr.

(define (find-master seq map-op predicate?)
  (define (find-master-helper remaining)
    (if (null? remaining) (error "Fail")
        (let ((master (car remaining)))
          (let ((mapped-seq (map (lambda (somearg) (map-op master somearg))
                                 seq)))
            (if (predicate? mapped-seq)
                master
                (find-master-helper (cdr remaining)))))))
  (find-master-helper seq))

;; Now we can define coerce and highest-type in terms of find-master

;; Could be faster if we don't use abstraction, but not as elegent. This is
;; mainly due to the fact that there's no clean to obtain the the master type of
;; a list of arguements without a some sort of tower data structure. This proves
;; the book's point that the `successive-raise` procedure makes apply-generic
;; sipmler. Another point is the fact that coerce args is not an "atmoic"
;; procedure, for it tries to find the master-type AND coerce all the
;; arguements. Unlike `successive-raise` which is atmoic. This means that
;; coerce is not as flexible, nor does it satisfy the unix philosophy.

(define (coerce args-list)
  (let ((master-type (find-master
                       (map type-tag args-list)
                       (lambda (x y)
                         (if (eq? x y) (lambda (a) a)
                             (get-coercion y x)))
                       has-null?))))
    (map (lambda (somearg) (get-coercion (type-tag somearg) master-type))
         args-list)))

(define (highest-type type-list)
  (find-master type-list
    (lambda (x y) (above? x y))
    (lambda (x) (not (has-false? x)))))

;; Since abstraciton is so magical, we can use for other things as well. For
;; instance to find the highest number in a list.
(define (highest-number numbers)
  (find-master numbers
    (lambda (master somearg) (>= master somearg))
    (lambda (x) (not (has-false? x)))))

;; This implelentaion of find-master is not as effcient as it could be, because
;; it runs the map-op on each element in the sequence, regardless of the
;; preceding sequence is any valid. However, since all other procedures we have
;; defined are sepereted by the find-master abstraction barrier, we can modify
;; find-master and everything will work out just fine.

```

周海青

```

;; 修改了meteorgan的程序, 当两个数一样时, 应该升级其中一个数
;; 以适应父级算法满 足子集, 直到没法再升级, 也可以避免循环调用
;; assuming that the types are arranged in a simple tower shape.
(define (apply-generic op . args)
  ;; raise s into t, if success, return s; else return #f
  (define (raise-into s t)
    (let ((s-type (type-tag s))
          (t-type (type-tag t)))
      (cond ((equal? s-type t-type)
              (if (get 'raise (list s-type))
                  ((get 'raise (list s-type)) (contents s))
                  #f
                  )
              )
             ((get 'raise (list s-type))
              (raise-into ((get 'raise (list s-type)) (contents s)) t)
              (else #f)))
           )
      )
    )
  )

```

```

(let ((type-tags (map type-tag args)))
  (let ((proc (get op type-tags)))
    (if proc
        (apply proc (map contents args))
        (if (= (length args) 2)
            (let ((a1 (car args))
                  (a2 (cadr args)))
              (cond
                ((raise-into a1 a2)
                 (apply-generic op (raise-into a1 a2) a2))
                ((raise-into a2 a1)
                 (apply-generic op a1 (raise-into a2 a1)))
                (else (error "No method for these types"
                            (list op type-tags))))))
            (error "No method for these types"
                   (list op type-tags)))))))

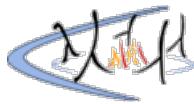
```

Masque

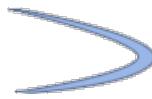
One of the problems or limitations of this level (or priority) table solution is that if we are going to add hundreds of different types and if we are to add one type in the midst of the tower, then the table maybe need to be updated to a large extent to accommodate this addition (imagine insert an element in an array, you have to move all elements after it one position to the right).

To maintain the additive principle of our modular design, one possible way is to make the table generated by itself.

Whenever we add a new type and its corresponding `raise` procedure to its direct supertype, we add to the table the result of comparison between this new type and all its supertypes. In this way, to add a new type, the only modification needed to existing modules is to change the `raise` procedure of this new type's direct subtype (it has to be changed anyway) and all the relations of comparison are established automatically and we don't have to worry about running out of level/priority numbers.



# sicp-ex-2.85



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.84) | Index | Next exercise (2.86) >>

meteorgan

```
;; add into rational package
(put 'project 'rational
     (lambda (x) (make-scheme-number (round (/ (numer x) (denom x))))))

;; add into real package
(put 'project 'real
     (lambda (x)
       (let ((rat (rationalize
                  (inexact->exact x) 1/100)))
         (make-rational
           (numerator rat)
           (denominator rat)))))

;; add into complex package
(put 'project 'complex
     (lambda (x) (make-real (real-part x)))))

(define (drop x)
  (let ((project-proc (get 'project (type-tag x))))
    (if project-proc
        (let ((project-number (project-proc (contents x))))
          (if (equ? project-number (raise project-number))
              (drop project-number)
              x))
        x)))

;; apply-generic
;; the only change is to apply drop to the (apply proc (map contents args))
(drop (apply proc (map contents args)))
```

Rptx

The previous solution did not work for me. Because I define raise in terms of apply-generic, so if I add drop to the result, it will enter an infinite loop. This also fails for equ? function, because you can't drop #t or #f.

```
; First, project, the procject package and drop.

(define (project arg)
  (apply-generic 'project arg))

(define (install-project-package)
  ;; internal procedures
  (define (complex->real x)
    (make-real (real-part x)))
  (define (real->integer x)
    (round x))
  (define (rational->integer x)
    (round (/ (car x) (cdr x))))
  ;; interface with system
  (put 'project '(complex)
       (lambda (x) (complex->real x)))
  (put 'project '(real)
       (lambda (x) (real->integer x)))
  (put 'project '(rational)
       (lambda (x) (rational->integer x)))
  'done)

(install-project-package)

(define (drop arg)
  (cond ((eq? (type-tag arg) 'scheme-number) arg)
        ((equ? arg (raise (project arg)))
         (drop (project arg)))
        (else arg)))

;; Here is my complete apply generic function. The change is the If statement
;; after proc. It tests to see if it is a 'raise operation, or an 'equ?
```

```

;; operation. If it is, is keeps the result as is, else it "drop"s it.

(define (apply-generic op . args)
  (define (higher-type types)
    (define (iter x types)
      (cond ((null? types) x)
            ((> (level x) (level (car types))) (iter x (cdr types)))
            (else (iter (car types) (cdr types))))))
    (if (null? (cdr types)) (car types) (iter (car types) (cdr types))))
  (define (raise-args args level-high-type)
    (define (iter-raise arg level-high-type)
      (if (< (level (type-tag arg)) level-high-type)
          (iter-raise (raise arg) level-high-type)
          arg))
    (map (lambda (arg) (iter-raise arg level-high-type)) args)))
  (define (not-all-same-type? lst)
    (define (loop lst)
      (cond ((null? lst) #f)
            ((eq? #f (car lst)) #t)
            (else (loop (cdr lst))))))
  (loop (map (lambda (x) (eq? (car lst) x))
             (cdr lst)))))

(let ((type-tags (map type-tag args)))
  (let ((proc (get op type-tags)))
    (if proc
        (let ((res (apply proc (map contents args))))
          (if (or (eq? op 'raise) (eq? op 'equ?)) res (drop res)))
        (if (not-all-same-type? type-tags)
            (let ((high-type (higher-type type-tags)))
              (let ((raised-args (raise-args args (level high-type))))
                (apply apply-generic (cons op raised-args))))
            (error
              "No Method for these types -- APPLY-GENERIC"
              (list op type-tags)))))))

```

Kaucher

The solution from Rptx looks like if there is a bug in his hierarchy. Hierarchy should be : integer -> rational -> real -> complex.

The solution looks like:

- integer -> rational
  - integer -> real -> complex
- A complex number cant be dropped to a rational number.

Kaucher

I think there is a bug in meteorgans solution. The line

```
(if (equ? project-number (raise project-number)))
```

Example: Drop the rational number 7/2.

- Apply 'project on 7/2 results in '4' "(round (/ 7 2))".
- Apply 'raise on 4 results in '4/1'. See 'raise from previous ex.:

```
(put 'raise 'integer (lambda (x) (make-rational x 1)))
```

Applying equ? on '4' and '4/1' returns #t, where '4/1' is definitely not equal to '7/2'. The raised object should be compared to the original object to check if the result of raising the dropped object equals the original. In Scheme:

```
(if (equ? (raise project-number) x))
```

Also: integers are ordinary numbers. So... a rational number can only be dropped to an integer if the denom is 1. For example: 7/1 can be dropped. 5/1 can be dropped. 1/3 can't be dropped. So you don't have to calculate the rounded value. Instead, just create an ordinary number with (numer x) and check if its raised value is the same as the original.

```
;;; rational package
(put 'project 'rational (lambda (x) (make-integer (numer x))))
```

Example:

- 7/2 => 7, (raise 7) => 7/1, (equ? 7/1 7/2) => #t
- 7/1 => 7, (raise 7) => 7/1, (equ? 7/1 7/1) => #t

Sphinxsky

First, not all results can be simplified. The results of arithmetic process can be simplified, but the results like raise and project can not be simplified, otherwise the results will be wrong. Therefore, in apply-generic, we should distinguish op parameters, which can be simplified by drop, and those can not.

Liskov

I was stuck in this exercise for a while, and I thought in a few possibilities to solve the problem, like using conditionals for raise and project, but know which op can be dropped shouldn't be responsibility of apply-generic. Another idea is to record in a table whether the result of the operation can be "dropped" or not.

I think that the best option is to abstract from apply-generic the procedure that really apply the op, to be used later by raise and project. Here is my try:

```
(define (apply-generic op . args)
  (define (handle-coercion types)
    (if (null? types)
        (error "No methods were found: APPLY_GENERIC" op args)
        (try-to-apply op (raise-args (car types)) handle-coercion (cdr types))))
  (define (raise-args type)
    (map
      (lambda (arg)
        (let ((raised-arg (raise-up-to arg type)))
          (if raised-arg raised-arg arg)))
      args))
  (define (raise-up-to obj type)
    (if (or (not obj) (equal? (type-tag obj) type))
        obj
        (raise-up-to
          (try-to-apply 'raise (list obj) identity false)
          type)))
  (let ((result
         (try-to-apply op args handle-coercion (map type-tag args))))
    (if (pair? result) (drop result) result)))
  (define (try-to-apply op args . callback)
    (let ((proc (get op (map type-tag args))))
      (cond (proc (apply proc (map contents args)))
            ((null? callback) '())
            (else (apply (car callback) (cdr callback)))))))
  (define (drop obj)
    (let ((projected-obj
           (try-to-apply 'project (list obj) identity false)))
      (if (and projected-obj (equ? projected-obj obj))
          (drop projected-obj)
          obj)))
  ;; raise and project can be rewritten as
  (define (raise x)
    (try-to-apply 'raise (list x) error "No methods were found: RAISE" x))
  (define (project x)
    (try-to-apply 'project (list x) error "No methods were found: PROJECT" x)))
```

# sicp-ex-2.86

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.85) | Index | Next exercise (2.87) >>

meteorgan

```
(define (sine x) (apply-generic 'sine x))
(define (cosine x) (apply-generic 'cosine x))

;; add into scheme-number package
(put 'sine 'scheme-number
     (lambda (x) (tag (sin x))))
(put 'cosine 'scheme-number
     (lambda (x) (tag (cos x)))))

;; add into rational package
(put 'sine 'rational
     (lambda (x) (tag (sin x))))
(put 'cosine 'rational
     (lambda (x) (tag (cos x)))))

;; To accomodate generic number in the complex package,
;; we should replace operators such as + , * with theirs
;; generic counterparts add, mul.
(define (add-complex z1 z2)
  (make-from-real-imag (add (real-part z1) (real-part z2))
                       (add (imag-part z1) (imag-part z2))))
(define (sub-complex z1 z2)
  (make-from-real-imag (sub (real-part z1) (real-part z2))
                       (sub (imag-part z1) (imag-part z2))))
(define (mul-complex z1 z2)
  (make-from-mag-ang (mul (magnitude z1) (magnitude z2))
                     (add (angle z1) (angle z2))))
(define (div-complex z1 z2)
  (make-from-mag-ang (div (magnitude z1) (magnitude z2))
                     (sub (angle z1) (angle z2))))
```

Kaucher

I think it should be

```
;; add into scheme-number package
(put 'sine 'rational (lambda (x) (sine (/ (numer x) (denom x)))))
(put 'cosine 'rational (lambda (x) (cosine (/ (numer x) (denom x)))))
```

fu7zed

but if we do that, it will become a real number, not a rational number anymore

...

(integer or rational sin/cos results are pretty rare)

shouldn't we raise it to real first, apply sine on the raised number, and put the sine function in the real-number package?

```
; in scheme-number package, will raise first to rational, then to real
(put 'sine 'scheme-number
      (lambda (x) (sine (raise (tag x)))))
; in rational package, will raise to real
(put 'sine 'rational
      (lambda (x) (sine (raise (tag x)))))
; in real package, apply `sin` primitive
(put 'sine 'real
      (lambda (x) (tag (sin x))))
```

yz

methods sine and cosine can only be defined on the scheme-number type, lower types have coercion int apply-generic. other methods like atan and expt(for the sqrt) should also be included, because the polar complex package need them.

```

;;; add into global
(define (sine x) (apply-generic 'sine x))
(define (cosine x) (apply-generic 'cosine x))
(define (arctan x) (apply-generic 'arctan x))
(define (exp x y) (apply-generic 'exp x y))

;;; add into rational package
(put 'sine '(number) (lambda (x) (tag (sin x))))
(put 'cosine '(number) (lambda (x) (tag (cos x))))
(put 'arctan '(number) (lambda (x) (tag (atan x))))
(put 'exp '(number number) (lambda (x y) (tag (expt x y)))))

;;; complex-rect package
(define (square x) (mul x x))
(define (sqrt x) (exp x 0.5))
(define (make-from-mag-ang r a) (cons (mul r (cosine a)) (mul r (sine a))))
(define (magnitude z) (sqrt (add (square (real-part z)) (square (imag-part z))))))
(define (angle z) (arctan (div (imag-part z) (real-part z)))))

;;; complex-polar package
(define (real-part z) (mul (magnitude z) (cosine (angle z))))
(define (imag-part z) (mul (magnitude z) (sine (angle z)))))

;;; complex package
(define (add-complex z1 z2)
  (make-from-real-imag (add (real-part z1) (real-part z2))
                       (add (imag-part z1) (imag-part z2))))
(define (sub-complex z1 z2)
  (make-from-real-imag (sub (real-part z1) (real-part z2))
                       (sub (imag-part z1) (imag-part z2))))
(define (mul-complex z1 z2)
  (make-from-mag-ang (mul (magnitude z1) (magnitude z2))
                     (add (angle z1) (angle z2))))
(define (div-complex z1 z2)
  (make-from-mag-ang (div (magnitude z1) (magnitude z2))
                     (sub (angle z1) (angle z2))))

```

Sphinxsky

My idea is to rewrite all basic operators to type operators through decorator, and to unify the input and output of all computing processes into scheme-number .

```

;; Converting data types to scheme-number
(define (install-type->scheme-number-package)
  ;; real -> scheme-number
  (put 'get-scheme-number '(real)
       (lambda (x) (make-scheme-number x)))

  ;; rational -> scheme-number
  (put 'get-scheme-number '(rational)
       (lambda (r)
         (make-scheme-number
          (contents (div (numer r) (denom r))))))

  ;; scheme-number -> scheme-number
  (put 'get-scheme-number '(scheme-number)
       (lambda (x) (make-scheme-number x)))

  'done)

;; Conversion interface
(define (get-scheme-number x)
  (apply-generic 'get-scheme-number x))

;; To rewrite basic operations into a form that can handle combined data
;; Return the result as scheme-number
(define (decorator f)
  ;; Unified input
  (define (transform args)
    (map
     (lambda (arg)
       (if (number? arg)
           (make-scheme-number arg)
           (get-scheme-number arg)))
     args))

  ;; Unified output
  (lambda (first . other)
    (make-scheme-number
     (let ((args (map
                  contents
                  (transform (cons first other)))))))
    (apply f args)))))


```

```
; To rewrite basic operations with decorator
;; You can also write like this:
;;      (set! + (decorator +))
;; So you don't have to change the code of complex package.
;; But once you do that, the other packages will suffer.
(define new-square (decorator square))
(define new-sqrt (decorator sqrt))
(define new-add (decorator +))
(define new-sub (decorator -))
(define new-mul (decorator *))
(define new-div (decorator /))
(define sine (decorator sin))
(define cosine (decorator cos))
(define new-atan (decorator atan))
```

If you don't want to change any source code, write this:

```
(define (decorator f)
  (define (transform args)
    (map
      (lambda (arg)
        (if (number? arg)
            (make-scheme-number arg)
            (get-scheme-number arg)))
      args))

  (lambda (first . other)
    (let ((arg-seg (cons first other)))
      (if (apply and (map number? arg-seg))
          (apply f arg-seg)
          (make-scheme-number
            (apply f (map contents (transform arg-seg)))))))))

(define (operator-overload!)
  (map
    (lambda (f)
      (set! f (decorator f)))
    (list square sqrt + - * / sin cos atan)))
```

partj

This exercise introduces the possibility that the various parts of a complex number (real, imaginary, magnitude, angle) might be data objects of various number types (ordinary scheme numbers, rational numbers, etc.).

This means that the various procedures of the complex number package that manipulate these parts will need to be generic over the supported number types. For example, in the procedure that gets the magnitude of a complex number in rectangular form, the procedures `square`, `+` and `sqrt` need to be generic:

```
(define (magnitude z)
  (sqrt (+ (square (real-part z))
            (square (imag-part z))))))
```

Below are the changes required in the system:

## 1. change the helper procedure definitions

```
(define (square x) (mul x x)) ; redirect to the generic procedure mul  
(define (sqroot x) (apply-generic 'sqroot x))  
(define (sine x) (apply-generic 'sine x))  
(define (cosine x) (apply-generic 'cosine x))  
(define (arctan v x) (apply-generic 'arctan v x))
```

2. implement the generic procedures for the various types

```

; add to rational package
(put 'sqrt 'rational (lambda (x)
    (make-rational (sqrt (numer x))
                  (sqrt (denom x)))))

; convert to ordinary numbers for the other procedures
(define (rational->scheme-number x)
  (make-scheme-number (/ (numer x) (denom x))))
(put 'sin 'rational (lambda (x)
    (sin (rational->scheme-number x))))
(put 'cos 'rational (lambda (x)
    (cos (rational->scheme-number x))))
(put 'arctan '(rational rational) (lambda (y x)
    (arctan (rational->scheme-number y)
            (rational->scheme-number x))))

```

3. replace the primitive procedures `+, -, *, /` by the generic procedures `add, sub, mul, div` in the rectangular, polar and the complex-number packages. e.g. in the polar package,

```

(define (real-part z) (mul (magnitude z) (cos (angle z))))
(define (imag-part z) (mul (magnitude z) (sin (angle z))))
(define (make-from-real-imag x y)
  (cons (sqrt (add (square x) (square y)))
        (arctan y x)))

```

Kaihao

partj's solution only considers the complex numbers whose real parts and imaginary parts are of the same type when doing `add` and `sub`, and the complex numbers whose magnitudes and angles are of the same type when doing `mul` and `div`.

For example, if `z1`'s real parts and imaginary parts are ordinary numbers, `z2`'s real parts and imaginary parts are rational numbers. Then `(add z1 z2)` will result an error.

So we need to add coercion.

```

(define (rational->scheme-number x)
  (let ((numer (car (contents x)))
        (denom (cdr (contents x))))
    (make-scheme-number (/ (* numer 1.0) denom)))

(put-coercion 'rational 'scheme-number rational->scheme-number)

```

Here is the full code, already tested in Racket.

```

#lang racket

;;
;;; put-coersion & get-coersion
;;; from https://gist.github.com/kinoshita-lab/b76a55759a0d0968cd97
;;;

(define coercion-list '())

(define (clear-coercion-list)
  (set! coercion-list '()))

(define (put-coercion type1 type2 item)
  (if (get-coercion type1 type2) coercion-list
      (set! coercion-list
            (cons (list type1 type2 item)
                  coercion-list)))

(define (get-coercion type1 type2)
  (define (get-type1 listItem)
    (car listItem))
  (define (get-type2 listItem)
    (cadr listItem))
  (define (get-item listItem)
    (caddr listItem))
  (define (get-coercion-iter list type1 type2)
    (if (null? list) #f
        (let ((top (car list)))
          (if (and (equal? type1 (get-type1 top))
                   (equal? type2 (get-type2 top)))
              (get-item top)
              (get-coercion-iter (cdr list) type1 type2)))))

(get-coercion-iter coercion-list type1 type2)

;;
;;; Put & Get, from https://stackoverflow.com/a/19114031
;;;

```

```

(define *op-table* (make-hash))
(define (put op type proc)
  (hash-set! *op-table* (list op type) proc))
(define (get op type)
  (hash-ref *op-table* (list op type) #f))

;;
;; Tags from 2.4.2
;;

(define (attach-tag type-tag z)
  (cons type-tag z))

(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "Not a pair: TYPE-TAG" datum)))
(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "Not a pair: CONTENT" datum)))

;;
;; 2.4.3 Data-Directed Programming and Additivity
;;

(define (install-rectangular-package)
  ;; internal procedures
  (define (real-part z) (car z))
  (define (imag-part z) (cdr z))
  (define (make-from-real-imag x y)
    (cons x y))

  ;; change sqrt, +, square, atan, *, cos, sin to generic procedures
  (define (magnitude z)
    (sqrt-generic (add (square-generic (real-part z))
                      (square-generic (imag-part z)))))

  (define (angle z)
    (atan-generic (imag-part z) (real-part z)))
  (define (make-from-mag-ang r a)
    (cons (mul r (cosine a)) (mul r (sine a)))))

  ;; interface to the rest of the system
  (define (tag x)
    (attach-tag 'rectangular x))
  (put 'real-part '(rectangular) real-part)
  (put 'imag-part '(rectangular) imag-part)
  (put 'magnitude '(rectangular) magnitude)
  (put 'angle '(rectangular) angle)
  (put 'make-from-real-imag 'rectangular
       (lambda (x y)
         (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'rectangular
       (lambda (r a)
         (tag (make-from-mag-ang r a))))
  'done)

(define (install-polar-package)
  ;; internal procedures
  (define (magnitude z) (car z))
  (define (angle z) (cdr z))
  (define (make-from-mag-ang r a) (cons r a))

  ;; change *, cos, sin, sqrt, +, square, atan to generic procedures
  (define (real-part z)
    (mul (magnitude z) (cosine (angle z))))
  (define (imag-part z)
    (mul (magnitude z) (sine (angle z))))
  (define (make-from-real-imag x y)
    (cons (sqrt-generic (add (square-generic x) (square-generic y)))
          (atan-generic y x)))

  ;; interface to the rest of the system
  (define (tag x) (attach-tag 'polar x))
  (put 'real-part '(polar) real-part)
  (put 'imag-part '(polar) imag-part)
  (put 'magnitude '(polar) magnitude)
  (put 'angle '(polar) angle)
  (put 'make-from-real-imag 'polar
       (lambda (x y)
         (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'polar
       (lambda (r a)
         (tag (make-from-mag-ang r a))))
  'done)

```

```

(define (real-part z)
  (apply-generic 'real-part z))
(define (imag-part z)
  (apply-generic 'imag-part z))
(define (magnitude z)
  (apply-generic 'magnitude z))
(define (angle z)
  (apply-generic 'angle z))

;;
;;; APPLY-GENERIC
;;; From 2.5.2 Combining Data of Different Types -> Coercion
;;;

(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (if (= (length args) 2)
              (let ((type1 (car type-tags))
                    (type2 (cadr type-tags))
                    (a1 (car args))
                    (a2 (cadr args)))
                (let ((t1->t2 (get-coercion type1 type2))
                      (t2->t1 (get-coercion type2 type1)))
                  (cond (t1->t2
                           (apply-generic op (t1->t2 a1) a2))
                        (t2->t1
                           (apply-generic op a1 (t2->t1 a2)))
                        (else (error "No method for these types:
                         APPLY-GENERIC"
                         (list op type-tags))))))
              (error "No method for these types: APPLY-GENERIC"
                     (list op type-tags)))))))

;;
;;; Added
;;; Coerce rational to scheme-number
;;;

(define (rational->scheme-number x)
  (let ((numer (car (contents x)))
        (denom (cdr (contents x))))
    (make-scheme-number (/ (* numer 1.0) denom)))))

(put-coercion 'rational 'scheme-number rational->scheme-number)

;;
;;; 2.5.1 Generic Arithmetic Operations
;;;

(define (add x y) (apply-generic 'add x y))
(define (sub x y) (apply-generic 'sub x y))
(define (mul x y) (apply-generic 'mul x y))
(define (div x y) (apply-generic 'div x y))

;; Add definitons of generic procedures
(define (sine x) (apply-generic 'sine x))
(define (cosine x) (apply-generic 'cosine x))
(define (sqrt-generic x) (apply-generic 'sqrt-generic x))
(define (atan-generic y x) (apply-generic 'atan-generic y x))
(define (square-generic x) (mul x x))

(define (install-scheme-number-package)
  (define (tag x)
    (attach-tag 'scheme-number x))
  (put 'add '(scheme-number scheme-number)
       (lambda (x y) (tag (+ x y))))
  (put 'sub '(scheme-number scheme-number)
       (lambda (x y) (tag (- x y))))
  (put 'mul '(scheme-number scheme-number)
       (lambda (x y) (tag (* x y))))
  (put 'div '(scheme-number scheme-number)
       (lambda (x y) (tag (/ x y))))
  (put 'make 'scheme-number
       (lambda (x) (tag x)))

  ; added
  (put 'sine '(scheme-number) (lambda (x) (tag (sin x))))
  (put 'cosine '(scheme-number) (lambda (x) (tag (cos x))))
  (put 'sqrt-generic '(scheme-number) (lambda (x) (tag (sqrt x))))
  (put 'atan-generic '(scheme-number scheme-number) (lambda (y x) (tag (atan y x)))))

  'done)

(define (make-scheme-number n)

```

```

((get 'make 'scheme-number) n))

(define (install-rational-package)
; internal procedures
(define (numer x) (car x))
(define (denom x) (cdr x))
(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g) (/ d g))))
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
             (* (denom x) (denom y))))
(define (sub-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (numer y) (denom x)))
             (* (denom x) (denom y))))
(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))
;; interface to rest of the system
(define (tag x) (attach-tag 'rational x))
(put 'add '(rational rational)
     (lambda (x y) (tag (add-rat x y))))
(put 'sub '(rational rational)
     (lambda (x y) (tag (sub-rat x y))))
(put 'mul '(rational rational)
     (lambda (x y) (tag (mul-rat x y))))
(put 'div '(rational rational)
     (lambda (x y) (tag (div-rat x y))))
(put 'make 'rational
     (lambda (n d) (tag (make-rat n d)))))

;; added
(define (tag-schemenumber x)
  (attach-tag 'scheme-number x))
(put 'sine '(rational)
     (lambda (x)
       (tag-schemenumber (sin (/ (numer x) (denom x))))))
(put 'cosine '(rational)
     (lambda (x)
       (tag-schemenumber (cos (/ (numer x) (denom x))))))
(put 'sqrt-generic '(rational)
     (lambda (x)
       (tag-schemenumber (sqrt (/ (* 1.0 (numer x)) (denom x)))))))
(put 'atan-generic '(rational rational)
     (lambda (y x)
       (tag-schemenumber (atan (/ (numer y) (denom y))
                               (/ (numer x) (denom x))))))

'done)

(define (make-rational n d)
  ((get 'make 'rational) n d))

(define (install-complex-package)
; imported procedures from rectangular
; and polar packages
(define (make-from-real-imag x y)
  ((get 'make-from-real-imag
        'rectangular)
   x y))
(define (make-from-mag-ang r a)
  ((get 'make-from-mag-ang 'polar)
   r a))
;; internal procedures

;; change +, -, *, / to generic procedures
(define (add-complex z1 z2)
  (make-from-real-imag
   (add (real-part z1) (real-part z2))
   (add (imag-part z1) (imag-part z2))))
(define (sub-complex z1 z2)
  (make-from-real-imag
   (sub (real-part z1) (real-part z2))
   (sub (imag-part z1) (imag-part z2))))
(define (mul-complex z1 z2)
  (make-from-mag-ang
   (mul (magnitude z1) (magnitude z2))
   (add (angle z1) (angle z2))))
(define (div-complex z1 z2)
  (make-from-mag-ang
   (div (magnitude z1) (magnitude z2))
   (sub (angle z1) (angle z2))))

```

```

;; interface to rest of the system
(define (tag z) (attach-tag 'complex z))
(put 'add '(complex complex)
     (lambda (z1 z2)
       (tag (add-complex z1 z2))))
(put 'sub '(complex complex)
     (lambda (z1 z2)
       (tag (sub-complex z1 z2))))
(put 'mul '(complex complex)
     (lambda (z1 z2)
       (tag (mul-complex z1 z2))))
(put 'div '(complex complex)
     (lambda (z1 z2)
       (tag (div-complex z1 z2))))
(put 'make-from-real-imag 'complex
     (lambda (x y)
       (tag (make-from-real-imag x y))))
(put 'make-from-mag-ang 'complex
     (lambda (r a)
       (tag (make-from-mag-ang r a))))
'done)

(define (make-complex-from-real-imag x y)
  ((get 'make-from-real-imag 'complex) x y))
(define (make-complex-from-mag-ang r a)
  ((get 'make-from-mag-ang 'complex) r a))

;;
;; Test
;;

(install-scheme-number-package)
(install-rational-package)
(install-rectangular-package)
(install-polar-package)
(install-complex-package)

(define x1 (make-scheme-number 1))
(define x2 (make-scheme-number 2))

(define y1 (make-rational 2 3))
(define y2 (make-rational 2 5))

(define z1 (make-complex-from-mag-ang x1 x2))
(define z2 (make-complex-from-mag-ang x1 y1))
(define z3 (make-complex-from-real-imag y1 y2))

(add z1 z2)
(add z1 z3)
(sub z1 z2)
(sub z1 z3)
(mul z1 z2)
(mul z1 z3)
(div z1 z2)
(div z1 z3)

```

# sicp-ex-2.87

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.86) | Index | Next exercise (2.88) >>

meterogan

```
; ; add into polynomial package
(define (zero-poly? poly)
  (define (zero-terms? termlist)
    (or (empty-termlist? termlist)
        (and (=zero? (coeff (first-term termlist)))
             (zero-terms? (rest-terms termlist))))))
  (zero-terms? (term-list poly)))
```

sam

Don't need zero-terms? because the polynomial representation in the book cannot have a term-list with zero terms (see definition for adjoin-term)

```
(define (zero-poly? p)
  (empty-termlist? (term-list p)))
```

Siki

The coefficient of a term can be either a number or a polynomial (Actually a number is a special polynomial with zero order). If it's number, we use the predicate ( $= x 0$ ); If it's represented as a polynomial, it cannot be zero. The polynomial representation we adopt does not contain zero terms.

```
; ; to be installed in polynomial package
(define (=zero? x)
  (define (poly? x)
    (pair? x))
  (cond ((number? x) (= x 0))
        ((poly? x) false)
        (else (error "Unknown type"))))

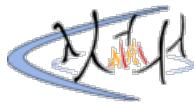
(put '=zero? 'polynomial =zero?)
```

master

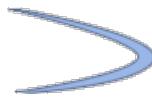
It's not necessary to do these tests, `=zero?` is a generic procedure which works on any type of argument, it will send the argument to the right place to do the comparison. Also, this procedure is not correct because it (falsely) assumes that there is no such thing as a polynomial with value zero. In fact, it is possible to have an arbitrarily long polynomial with all zero coefficients. It will be simplified to zero, sure, but we don't know whether the input is simplified or not. I think the following procedure works (can't test it):

```
(define (=zero? poly)
  (if (empty-termlist? poly)
      (the-empty-termlist)
      (if (=zero? (coeff (first-term poly)))
          (=zero? (rest-terms poly))
          #f)))
```

If it encounters any non-zero coefficient then the polynomial is obviously not equal to zero.



# sicp-ex-2.88



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.87) | Index | Next exercise (2.89) >>

meterogan

```
(define (negate x) (apply-generic 'negate x))

;; add into scheme-number package
(put 'negate 'scheme-number
     (lambda (n) (tag (- n)))))

;; add into rational package
(put 'negate 'rational
     (lambda (rat) (make-rational (- (numer rat)) (denom rat)))))

;; add into complex package
(put 'negate 'complex
     (lambda (z) (make-from-real-imag (- (real-part z))
                                       (- (imag-part z))))))

;; add into polynomial package
(define (negate-terms termlist)
  (if (empty-termlist? termlist)
      the-empty-termlist
      (let ((t (first-term termlist)))
        (adjoin-term (make-term (order t) (negate (coeff t)))
                     (negate-terms (rest-terms termlist))))))
  (put 'negate 'polynomial
       (lambda (poly) (make-polynomial (variable poly)
                                         (negate-terms (term-list poly)))))

  (put 'sub '(polynomial polynomial)
       (lambda (x y) (tag (add-poly x (negate y))))))
```

Kaihao

I have fixed all the errors in meterogan's solution.

```
(define (negate x) (apply-generic 'negate x))

;; add into scheme-number package
(put 'negate '(scheme-number)
     (lambda (x) (tag (- x)))))

;; add into rational package
(put 'negate '(rational)
     (lambda (x) (tag (make-rat (- (numer x)) (denom x)))))

;; add into complex package
(put 'negate '(complex)
     (lambda (z) (tag (make-from-real-imag (- (real-part z))
                                           (- (imag-part z))))))

;; add into polynomial package
(define (negate-terms termlist)
  (if (empty-termlist? termlist)
      the-empty-termlist
      (let ((t (first-term termlist)))
        (adjoin-term (make-term (order t) (negate (coeff t)))
                     (negate-terms (rest-terms termlist))))))
  (define (negate-poly p)
    (make-poly (variable p) (negate-terms (term-list p))))
  (define (sub-poly p1 p2)
    (if (same-variable? (variable p1) (variable p2))
        (add-poly p1 (negate-poly p2))
        (error "Polys not in the same var: SUB-POLY"
              (list p1 p2))))
  (put 'negate '(polynomial)
       (lambda (p) (tag (negate-poly p))))
  (put 'sub '(polynomial polynomial)
       (lambda (p1 p2) (tag (sub-poly p1 p2)))))
```

```

hi-artem define (negate-terms termlist)
  (map
    (lambda (t) (make-term (order t)
                           (- (coeff t))))
  termlist))

```

CrazyAlvaro *; I got the same idea as hi-artem to use map procedure, however I think we need to use the generic negate for coeff as well*

```

(define (negate-poly p)
  (make-polynomial (variable p)
    (map
      (lambda (term)
        (make-term
          (order term)
          (negate (coeff term))))
      (term-list p))))

```

Sphinxsky I think hi-artem is wrong and meterogan is right. Because MAP only works on lists, it is impossible to achieve Abstract masking if MAP is used. However, there are also errors in meterogan's writing. It should look like this:

```

;; add into complex package
(put 'negate 'complex
  (lambda (z) (make-from-real-imag (negate (real-part z))
                                    (negate (imag-part z)))))

(put 'sub '(polynomial polynomial)
  (lambda (x y) (tag (add-poly x (contents (negate (tag y)))))))

```

Totti I didn't follow the book's suggestion and implemented the sub procedure by adding the terms of the first polynomial to the multiplication of all terms of the second by -1 or  $-1x^0$ .

```

(define (sub-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
    (make-poly (variable p1)
      (add-terms (term-list p1)
        (mul-term-by-all-terms (make-term 0 -1)
          (term-list p2))))
    (error "Polys not in same var -- SUB-POLY"
      (list p1 p2))))

```

Alice Other solutions seems to forget to tag their constructors.

```

;; generic negation
(define (negate x)
  (apply-generic 'negate x))

;; in scheme number package
(put 'negate 'scheme-number (lambda (x) (tag (- x)))) ; tag is optional here

;; rat package
(put 'negate 'rational (lambda (x) (tag (make-rat (- (numer x)) (denom x)))))

;; into rectangular
(put 'negate 'rectangular (lambda (x)
  (tag (make-from-real-imag
    (- (real-part x)) (- (imag-part x))))))

;; into polar
(put 'negate 'polar (lambda (x) (tag (make-from-mag-ang
  (mag x) (+ (ang x) 180)))))

;; into complex
(put 'negate 'complex negate) ; double tag system

;; into poly

```

```

(define (negate-terms L)
  (if (empty-termlist? L) the-empty-term-list ; what a stupid name, why "the"?
      (let ((t1 (first-term L)))
        (let ((negated-t1
              (make-term (order t1) (negate (coeff t1)))))
          (adjoin-term negated-t1
                      (negate-terms (rest L))))))
  ;; Silly, but fun alternative. I'm assuming we can always multiply by minus-one
(define (negate-terms poly)
  (let ((minus-one (make-term 0 -1)))
    (make-polynomial (var poly)
                     (mul-term-by-all-terms minus-one (terms poly)))))
(define (negate-poly poly)
  (make-poly (var poly)
             (negat-terms (terms poly))))
(put 'negate 'polynomial (lambda (x) (tag (negate-poly x)))))

;; generic sub in terms of negate and add
(define (sub x y)
  (apply-generic 'add x (negate y)))

;; We can subtract polynomials now :D

```

master

Can't we just copy the strategy used in add-terms? Subtraction of polynomials boils down to subtraction of its coefficients, for which sub is defined on all types up to this point. Not that having a unary negation procedure isn't valuable, but it doesn't really seem strictly necessary in the context of the exercise. In the spirit of the book, we should really abstract out all the common code, but that's another issue.

```

(define (sub-terms L1 L2)
  (cond ((empty-termlist? L1) L2)
        ((empty-termlist? L2) L1)
        (else
         (let ((t1 (first-term L1))
               (t2 (first-term L2)))
           (cond ((> (order t1) (order t2))
                  (adjoin-term
                   t1 (sub-terms (rest-terms t1) L2)))
                 ((< (order t1) (order t2))
                  (adjoin-term
                   t2 (sub-terms L1 (rest-terms L2))))
                 (else
                  (adjoin-term
                   (make-term (order t1)
                             (sub (coeff t1) (coeff t2)))
                   (sub-terms (rest-terms L1)
                             (rest-terms L2)))))))))

```

# sicp-ex-2.89

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.88) | Index | Next exercise (2.90) >>

meteorgan ;; all we should change are procedure first-term and adjoin-term

```
(define (first-term term-list)
  (list (car term-list) (- (length term-list) 1)))
(define (adjoin-term term term-list)
  (let ((exponent (order term))
        (len (length term-list)))
    (define (iter-adjoin times terms)
      (cond ((=zero? (coeff term))
              terms)
            ((= exponent times)
             (cons (coeff term) terms))
            (else (iter-adjoin (+ times 1)
                               (cons 0 terms))))))
    (iter-adjoin len term-list)))
```

hattivat ;; an even shorter solution to achieve the same result

```
(define (first-term term-list)
  (make-term (- (len term-list) 1) (car term-list)))

(define (adjoin-term term term-list)
  (cond ((=zero? term) term-list)
        ((equ? (order term) (length term-list)) (cons (coeff term) term-list))
        (else (adjoin-term term (cons 0 term-list)))))
```

Kaihao In adjoin-term, we need to use =zero? on (coeff term), not on term. And equ? is not necessary because we are comparing just two ordinary numbers.

Here is the fixed code:

```
(define (first-term term-list)
  (make-term (- (len term-list) 1) (car term-list)))

(define (adjoin-term term term-list)
  (cond ((=zero? (coeff term)) term-list)
        ((= (order term) (length term-list)) (cons (coeff term) term-list))
        (else (adjoin-term term (cons 0 term-list)))))
```

Marisa

We don't need to change much. Also, since adjoin-term is properly used in add-term and mul-term, I don't think we have to change it drastically like some people did.

;; List the term, and find the order

```
(define (first-term term-list)
  (list
    (car term-list) ; first term
    (- 1 (length term-list))) ; order is length-1 (we index polynomials from 0)

;; Slightly modified to cons the coeff instead of the actual term
(define (adjoin-term term term-list)
  (let ((coeff-of-term (coeff term)))
    (if (=zero? coeff-of-term)
        term-list
        (cons coeff-of-term term-list))))
```

felixm

It is probably me, but none of the solution above worked for me.

meteorgan's solution has bracketing errors and even after fixing them there seems to be logical errors.

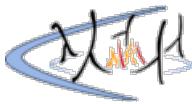
hattivat uses =zero? directly on term which does not work because add-terms calls adjoin-term on regular terms, i.e. one that have coeff and order.

Finally, Marisa's solution runs, but creates false results for my tests.

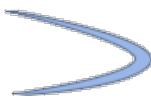
The following code worked for me when I added it to the poly-package from the book. Note that not considering zero terms would yield invalid results for multiplication.

```
(define (first-term term-list)
  (make-term (- (length term-list) 1)
             (car term-list)))

(define (adjoin-term term term-list)
  (let ((coeff-term (coeff term))
        (order-term (order term))
        (length-terms (length term-list)))
    (cond
      ((= order-term length-terms) (cons coeff-term term-list))
      ((< order-term length-terms) (error "Cannot adjoin lower-order term to terms"))
      (else (cons coeff-term (adjoin-term (make-term (- order-term 1) 0) term-list))))))
```



# sicp-ex-2.90



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.89) | Index | Next exercise (2.91) >>

woofy

```
; install different term-list representations correspondingly:  
  
; sparse as in the text  
(define (install-dense-term-list)  
  
  ;inner  
  (define (adjoin-term term term-list)  
    (if (=zero? (coeff term))  
        term-list  
        (cons (term term-list))))  
  
  (define (first-term term-list) (car term-list))  
  (define (rest-terms term-list) (cdr term-list))  
  
  ;interface  
  (define (tag term-list) (attach-tag 'sparse term-list))  
  
  (put 'adjoin-term 'sparse adjoin-term)  
  
  (put 'first-term '(sparse)  
       (lambda (term-list) (first-term term-list)))  
  
  (put 'rest-term '(sparse)  
       (lambda (term-list) (tag (rest-terms term-list))))  
  'done)  
  
; dense from ex_2.89  
(define (install-dense-term-list)  
  
  ;inner  
  (define (adjoin-term term term-list)  
    (cond ((=zero? (coeff term)) term-list)  
          ((=equ? (order term) (length term-list)) (cons (coeff term) term-list))  
          (else (adjoin-term term (cons 0 term-list)))))  
  
  (define (first-term term-list) (make-term (- (length term-list) 1) (car term-list)))  
  (define (rest-terms term-list) (cdr term-list))  
  
  ;interface  
  (define (tag term-list) (attach-tag 'dense term-list))  
  
  (put 'adjoin-term 'dense adjoin-term)  
  
  (put 'first-term '(dense)  
       (lambda (term-list) (first-term term-list)))  
  
  (put 'rest-term '(dense)  
       (lambda (term-list) (tag (rest-terms term-list))))  
  'done)  
  
; since term representation is all the same,  
; we only registered the term-list type into the table.  
(define (adjoin-term term term-list)  
  ((get 'adjoin-term (type-tag term-list)) term term-list))  
  
(define (first-term term-list) (apply-generic 'first-term term-list))  
(define (rest-term term-list) (apply-generic 'rest-term term-list))
```

Kaihao

We only need to change how terms and term lists are represented. All terms are tagged "term", and all term lists are tagged "sparse" or "dense".

```
(define (install-terms-package)  
  ;; internal procedures  
  
  ;; not changed
```

```

(define (the-empty-term-list) '())
(define (rest-terms term-list) (cdr term-list))
(define (empty-term-list? term-list)
  (null? term-list))
(define (make-term order coeff)
  (list order coeff))
(define (order term) (car term))
(define (coeff term) (cadr term))

;; sparse representation
(define (adjoin-term-sparse term term-list)
  (if (=zero? (coeff term))
      term-list
      (cons term term-list)))
(define (first-term-sparse term-list) (car term-list))
;; dense representation, from Exercise 2.89
(define (first-term-dense term-list)
  (make-term (- (length term-list) 1)
            (car term-list)))
(define (adjoin-term-dense term term-list)
  (cond ((=zero? (coeff term)) term-list)
        ((= (order term) (length (term-list)))
         (cons (coeff term) term-list))
        (else (adjoin-term-dense term (cons 0 term-list)))))

;; interface to the rest of the system
(define (tag-sparse x) (attach-tag 'sparse x))
(define (tag-dense x) (attach-tag 'dense x))
(define (tag-term x) (attach-tag 'term x))
(put 'adjoin-term '(term sparse)
     (lambda (term term-list)
       (tag-sparse (adjoin-term-sparse term term-list))))
(put 'adjoin-term '(term dense)
     (lambda (term term-list)
       (tag-dense (adjoin-term-dense term term-list))))
(put 'the-empty-term-list 'sparse
     (lambda () (tag-sparse (the-empty-term-list))))
(put 'the-empty-term-list 'dense
     (lambda () (tag-dense (the-empty-term-list))))
(put 'first-term '(sparse)
     (lambda (term-list)
       (tag-term (first-term-sparse term-list))))
(put 'first-term '(dense)
     (lambda (term-list)
       (tag-term (first-term-dense term-list))))
(put 'rest-terms '(sparse)
     (lambda (term-list)
       (tag-sparse (rest-terms term-list))))
(put 'rest-terms '(dense)
     (lambda (term-list)
       (tag-dense (rest-terms term-list))))
(put 'empty-term-list? '(sparse)
     (lambda (term-list) (empty-term-list? term-list)))
(put 'empty-term-list? '(dense)
     (lambda (term-list) (empty-term-list? term-list)))
(put 'make-term 'term
     (lambda (order coeff)
       (tag-term (make-term order coeff))))
(put 'order '(term) order)
(put 'coeff '(term) coeff)
'done)

(define (adjoin-term term term-list)
  (apply-generic 'adjoin-term term term-list))
(define (first-term term-list)
  (apply-generic 'first-term term-list))
(define (rest-terms term-list)
  (apply-generic 'rest-terms term-list))
(define (empty-term-list? term-list)
  (apply-generic 'empty-term-list? term-list))
(define (order term)
  (apply-generic 'order term))
(define (coeff term)
  (apply-generic 'coeff term))

(define make-sparse-empty-term-list
  (get 'the-empty-term-list 'sparse))
(define make-dense-empty-term-list
  (get 'the-empty-term-list 'dense))
(define (make-term order coeff)
  ((get 'make-term 'term) order coeff))

;; Define the-empty-term-list used in mul-terms
;; and mul-term-by-all-terms.
;; Here we use make-sparse-empty-list, but
;; make-dense-empty-termlist is also fine.

```

```
(define the-empty-term-list make-sparse-empty-list)
```

Rptx

This system can even do operations on different types of polynomials. `adjoin-term` is not perfect. It can't adjoin a term in the middle of a polynomial correctly. I will fix it later.

```
; these have to be added to the polynomial package. So now it dispatches to
; the generic procedures.

(define (the-empty-term-list term-list)
  (let ((proc (get 'the-empty-term-list (type-tag term-list))))
    (if proc
        (proc)
        (error "No proc found -- THE-EMPTY-TERMLIST" term-list)))
  (define (rest-terms term-list)
    (let ((proc (get 'rest-terms (type-tag term-list))))
      (if proc
          (proc term-list)
          (error "-- REST-TERMS" term-list)))
  (define (empty-term-list? term-list)
    (let ((proc (get 'empty-term-list? (type-tag term-list))))
      (if proc
          (proc term-list)
          (error "-- EMPTY-TERMLIST?" term-list)))
  (define (make-term order coeff) (list order coeff))
  (define (order term)
    (if (pair? term)
        (car term)
        (error "Term not pair -- ORDER" term)))
  (define (coeff term)
    (if (pair? term)
        (cadr term)
        (error "Term not pair -- COEFF" term)))

; here is the term-list package, which has the constructors and selectors for
; the dense and sparse polynomials.

; the generic first-term procedure.
(define (first-term term-list)
  (let ((proc (get 'first-term (type-tag term-list))))
    (if proc
        (proc term-list)
        (error "No first-term for this list -- FIRST-TERM" term-list)))

; the package with the constructors, selectors, and other helper procedures
; I had to implement.
(define (install-polynomial-term-package)
  (define (first-term-dense term-list)
    (if (empty-term-list? term-list)
        '()
        (list
         (- (length (cdr term-list)) 1)
         (car (cdr term-list)))))
  (define (first-term-sparse term-list)
    (if (empty-term-list? term-list)
        '()
        (cadr term-list)))
  (define (prep-term-dense term)
    (if (null? term)
        '()
        (cdr term))) ; -> only the coeff for a dense term-list
  (define (prep-term-sparse term)
    (if (null? term)
        '()
        (list term))) ; -> (order coeff) for a sparse term-list
  (define (the-empty-term-list-dense) '(dense))
  (define (the-empty-term-list-sparse) '(sparse))
  (define (rest-terms term-list) (cons (type-tag term-list) (cddr term-list)))
  (define (empty-term-list? term-list)
    (if (pair? term-list)
        (>= 1 (length term-list))
        (error "Term-list not pair -- EMPTY-TERMLIST?" term-list)))
  (define (make-polynomial-dense var terms)
    (make-polynomial var (cons 'dense (map cadr terms))))
  (define (make-polynomial-sparse var terms)
    (make-polynomial var (cons 'sparse terms)))
  (put 'first-term 'sparse
       (lambda (term-list) (first-term-sparse term-list)))
  (put 'first-term 'dense
       (lambda (term-list) (first-term-dense term-list)))
  (put 'prep-term 'dense
       (lambda (term) (prep-term-dense term))))
```

```

(put 'prep-term 'sparse
     (lambda (term) (prep-term-sparse term)))
(put 'rest-terms 'dense
     (lambda (term-list) (rest-terms term-list)))
(put 'rest-terms 'sparse
     (lambda (term-list) (rest-terms term-list)))
(put 'empty-term-list? 'dense
     (lambda (term-list) (empty-term-list? term-list)))
(put 'empty-term-list? 'sparse
     (lambda (term-list) (empty-term-list? term-list)))
(put 'the-empty-term-list 'dense
     (lambda () (the-empty-term-list-dense)))
(put 'the-empty-term-list 'sparse
     (lambda () (the-empty-term-list-sparse)))
(put 'make-polynomial 'sparse
     (lambda (var terms) (make-polynomial-sparse var terms)))
(put 'make-polynomial 'dense
     (lambda (var terms) (make-polynomial-dense var terms)))
'done)

(install-polynomial-term-package)

; I had to change the adjoin-term procedure. It now does
; zero padding so we can `mul` dense polynomials correctly.

(define (zero-pad x type)
  (if (eq? type 'sparse)
      '()
      (if (= x 0)
          '()
          (cons 0 (add-zeros (- x 1))))))

(define (adjoin-term term term-list)
  (let ((prep-term ((get 'prep-term (type-tag term-list)) term))
        (prep-first-term ((get 'prep-term (type-tag term-list))
                          (first-term term-list))))
        (cond ((=zero? (coeff term)) term-list)
              ((empty-term-list? term-list) (append (the-empty-term-list term-list)
                                                    prep-term
                                                    (zero-pad (order term)
                                                          (type-tag
                                                           term-list))))
              ((> (order term) (order (first-term term-list)))
               (append (list (car term-list))
                      prep-term
                      (zero-pad (- (- (order term)
                                    (order (first-term term-list)))
                                   1) (type-tag term-list)))
                      (cdr term-list)))
              (else
                (append prep-first-term
                        (adjoin-term term (rest-terms term-list)))))))
    ; here is `negate` now it creates a polynomial of the correct
    ; type

  (define (negate p)
    (let ((neg-p ((get 'make-polynomial (type-tag (term-list p)))
                  (variable p) (list (make-term 0 -1)))))
      (mul-poly (cdr neg-p) p)))

```

Paper

My solution is perhaps not as complete overall but I feel it is closer to what the authors would have expected since its approach is copied from section 2.4.3.

```

(define (install-polynomial-dense-package)
  (define (adjoin-term term term-list)
    (if (= (order term) (length term-list))
        (cons (coeff term) term-list)
        (adjoin-term term
                    (cons 0 term-list))))
  (define (first-term term-list)
    (make-term (car term-list)
               (- (length term-list) 1)))
  ; here place all the other procedures
  (define (tag x)
    (attach-tag 'polynomial-dense x))
  (put 'adjoin-term 'polynomial-dense
       (lambda (term term-list) (tag (adjoin-term term term-list))))
  (put 'first-term 'polynomial-dense
       (lambda (term-list) (tag (first-term term-list))))
  ; here place all the other calls to put
  'done)

  (define (install-polynomial-sparse-package)

```

```

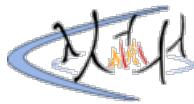
(define (adjoin-term term term-list)
  (if (=zero? (coeff term))
      term-list
      (cons term term-list)))
(define (first-term term-list) (car term-list))
; here place all the other procedures
(define (tag x)
  (attach-tag 'polynomial-sparse x))
(put 'adjoin-term 'polynomial-sparse
     (lambda (term term-list) (tag (adjoin-term term term-list))))
(put 'first-term 'polynomial-sparse
     (lambda (term-list) (tag (first-term term-list))))
; here place all the other calls to put
'done)

; committed repeated procedures -- they would be the same for both packages

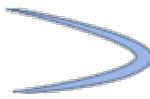
(define (make-poly variable term-list)
  (if (poly-sparse? term-list)
      ((get 'make-poly 'polynomial-sparse) term-list)
      ((get 'make-poly 'polynomial-dense) term-list)))

(define (poly-sparse? term-list)
; a polynomial is sparse if at least half of its coefficients are zero
(define (iter zeros remaining)
  (cond ((null? remaining)
         zeros)
        ((equ? (coeff (car remaining)) 0)
         (iter (+ zeros 1) (cdr remaining)))
        (else
         (iter zeros (cdr remaining)))))
; the above could also be done with reduce
(> (iter 0 term-list)
    (/ (length term-list) 2)))

```



# sicp-ex-2.91



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (2.90) | Index | Next exercise (2.92) >>

```
(define (div-terms L1 L2)
  (if (empty-term-list? L1)
      (list (the-empty-term-list) (the-empty-term-list))
      (let ((t1 (first-term L1))
            (t2 (first-term L2)))
        (if (> (order t2) (order t1))
            (list (the-empty-term-list) L1)
            (let ((new-c (div (coeff t1) (coeff t2)))
                  (new-o (- (order t1) (order t2))))
              (let ((rest-of-result
                     (div-terms (sub-terms L1
                                           (mul-term-by-all-terms
                                            (make-term new-o new-c)
                                            L2))
                               L2)))
                (list (adjoin-term (make-term new-o new-c)
                                   (first rest-of-result))
                      (second rest-of-result))))))))
  (define (div-poly p1 p2)
    (if (same-variable? (variable p1) (variable p2))
        (map (lambda (term-list) (make-poly (variable p1) term-list))
              (div-terms (term-list p1) (term-list p2)))
        (error "Polys not in same var -- DIV-POLY"
              (list p1 p2))))
```

woofy

```
(define (div-terms L1 L2)
  (if (empty-term-list? L1)
      (list (the-empty-term-list) (the-empty-term-list))
      (let ((t1 (first-term L1))
            (t2 (first-term L2)))
        (if (> (order t2) (order t1))
            (list (the-empty-term-list) L1)
            (let ((new-c (div (coeff t1) (coeff t2)))
                  (new-o (- (order t1) (order t2))))
              (let ((rest-of-result
                     (div-terms
                       (sub-terms (
                         L1
                         (mul-term-by-all-terms (make-term new-o new-c) L1)))
                     L2)))
                (list (adjoin-term (make-term new-o new-c) (car rest-of-result))
                      (cadr rest-of-result)))))))
  (define (div-poly p1 p2)
    (if (same-variable? (variable p1) (variable p2))
        (let ((term-div-result (div-terms (term-list p1) (term-list p2))))
          (let ((quotient-term-list (car term-div-result)
                (remainder-term-list (cadr term-dev-result))))
            (list (make-poly (variable p1) quotient-term-list)
                  (make-poly (variable p1) remainder-term-list))))
        (error "Polys not in same var -- DIV-POLY" (list p1 p2))))
```

meteorgan

```
(define (div-terms L1 L2)
  (if (empty-term-list? L1)
      (list (the-empty-term-list) (the-empty-term-list))
      (let ((t1 (first-term L1))
            (t2 (first-term L2)))
        (if (> (order t2) (order t1))
            (list (the-empty-term-list) L1)
```

```

(let ((new-c (div (coeff t1) (coeff t2)))
      (new-o (- (order t1) (order t2)))
      (new-t (make-term new-o new-c)))
  (let ((rest-of-result
         (div-terms (add-poly L1 (negate (mul-poly (list new-t) L2)))
                    L2)))
    (list (adjoin-term new-t
                       (car rest-of-result))
          (cadr rest-of-result)))))

(define (div-poly poly1 poly2)
  (if (same-variable? (variable poly1) (variable poly2))
      (make-poly (variable poly1)
                 (div-terms (term-list poly1)
                            (term-list poly2)))
      (error "not the same variable" (list poly1 poly2))))

```

Siki

The answer above has one inaccurate point: add-terms & mul-terms should be used instead of add-poly & mul-poly in the first paragraph.

```

(define (div-terms L1 L2)
  (if (empty-termlist? L1)
      (list (the-empty-termlist) (the-empty-termlist))
      (let ((t1 (first-term L1))
            (t2 (first-term L2)))
        (if (> (order t2) (order t1))
            (list (the-empty-termlist) L1)
            (let ((new-c (div (coeff t1) (coeff t2)))
                  (new-o (- (order t1) (order t2))))
              (let ((rest-of-result
                     (div-terms (sub-terms L1
                                           (mul-terms L2
                                                      (list (make-term new-o
                                                                      new-c)))))))
                (list (adjoin-term (make-term new-o new-c)
                                   (car rest-of-result))
                      (cadr rest-of-result)))))))

```

Gera

Here's an answer that modifies the given code a bit to be easier to work with, and makes use of the mul-term-by-all-terms procedure. Also a corrected div-poly procedure, since the one in the first post is analogous to add-poly, when it shouldn't be since div-terms returns a list as should div-poly.

```

(define (div-terms l1 l2)
  (if (empty-termlist? l1)
      (list (the-empty-termlist) (the-empty-termlist))
      (let ((t1 (first-term l1))
            (t2 (first-term l2)))
        (if (> (order t2) (order t1))
            (list (the-empty-termlist) l1)
            (let ((first-term (make-term (- (order t1) (order t2))
                                         (div (coeff t1) (coeff t2)))))
              (let ((rest-of-result
                     (div-terms (sub-terms l1
                                           (mul-term-by-all-terms first-term
                                                               l2)))
                     (12)))
                (list (adjoin-term first-term (car rest-of-result))
                      (cadr rest-of-result)))))))

(define (div-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (let ((results (div-terms (term-list p1)
                                (term-list p2))))
        (list (make-poly (variable p1) (car results)) (cadr results)))
      (error "Polys not in same var -- div-poly"
            (list p1 p2))))

```

Shawn

All the solutions above missed the point that the results of div-poly are two polynomials. What they returns are a quotient poly and a remainder term list. When we return the result, we should make two polynomials, not one. The correct implementation for div-poly should be:

```

(define (div-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (let ((result (div-terms (term-list p1)
                               (term-list p2))))

```

```

(list (make-poly (variable p1)
                  (car result))
      (make-poly (variable p1)
                  (cadr result)))
  (error "Variable is not the same -- DIV-POLY" (list (variable p1) (variable
p2))))))

```

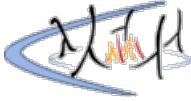
anonymous

My div-terms is similar to the one of Gera but I use the fact that the first term of L1 is canceled by the new term multiplied by the first term of L2 so we don't have to touch the first term of L1 and L2 when we do the subtraction. Doing this we can save one multiplication of two terms. Here is the little improvement:

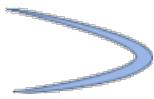
```

(define (div-terms L1 L2)
  (if (empty-termlist? L1)
      (list (the-empty-termlist) (the-empty-termlist))
      (let ((t1 (first-term L1))
            (t2 (first-term L2)))
        (if (> (order t2) (order t1))
            (list (the-empty-termlist) L1)
            (let ((new-c (div (coeff t1) (coeff t2)))
                  (new-o (- (order t1) (order t2))))
              (let ((rest-of-result
                     (div-terms (sub-terms (rest-terms L1) ; only the rest terms of
L1
                     (mul-term-by-all-terms
                      (make-term new-o new-c)
                      (rest-terms L2))) ; only the rest
terms of L2
                     L2)))
                (list (adjoin-term (make-term new-o new-c) (car rest-of-result))
                      (cadr rest-of-result)))))))

```



# sicp-ex-2.92



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

[<< Previous exercise \(2.91\) | Index | Next exercise \(2.93\) >>](#)

Rptx Here is my answer. It gives x a value of 1, and all other variables are 0. So everything will be "raised" to a polynomial in x. I have also implemented mixed operation of polynomials with all the other types used in the chapter.

```

(define (install-polynomial-package)
  ;; internal procedures
  ;; representation of poly
  (define (make-poly variable term-list)
    (cons variable term-list))
  (define (polynomial? p)
    (eq? 'polynomial (car p)))
  (define (variable p) (car p))
  (define (term-list p) (cdr p))
  (define (variable? x)
    (symbol? x))
  (define (same-variable? x y)
    (and (variable? x) (variable? y) (eq? x y)))

  ;; representation of terms and term lists
  (define (add-poly p1 p2)
    (display "var p1 ") (display p1) (newline)
    (display "var p2 ") (display p2) (newline)
    (if (same-variable? (variable p1) (variable p2))
        (make-poly (variable p1)
                   (add-terms (term-list p1)
                              (term-list p2)))
        (let ((ordered-polys (order-polys p1 p2)))
          (let ((high-p (higher-order-poly ordered-polys))
                (low-p (lower-order-poly ordered-polys)))
            (let ((raised-p (change-poly-var low-p)))
              (if (same-variable? (variable high-p)
                                 (variable (cdr raised-p)))
                  (add-poly high-p (cdr raised-p)) ;-> cdr for 'polynomial. Should fix
this,
here.
Poly"
                  (error "Poly not in same variable, and can't change either! -- ADD-
POLY"
                         (list high-p (cdr raised-p))))))))
  (define (add-terms L1 L2)
    (cond ((empty-term-list? L1) L2)
          ((empty-term-list? L2) L1)
          (else
            (let ((t1 (first-term L1))
                  (t2 (first-term L2)))
              (cond ((> (order t1) (order t2))
                     (adjoin-term
                      t1 (add-terms (rest-terms L1) L2)))
                    ((< (order t1) (order t2))
                     (adjoin-term
                      t2 (add-terms L1 (rest-terms L2))))
                    (else
                      (adjoin-term
                        (make-term (order t1)
                                   (add (coeff t1) (coeff t2)))
                        (add-terms (rest-terms L1)
                                   (rest-terms L2))))))))))

  (define (mul-poly p1 p2)
    (if (same-variable? (variable p1) (variable p2))
        (make-poly (variable p1)
                   (mul-terms (term-list p1)
                              (term-list p2)))
        (let ((ordered-polys (order-polys p1 p2)))
          (let ((high-p (higher-order-poly ordered-polys))
                (low-p (lower-order-poly ordered-polys)))
            (let ((raised-p (change-poly-var low-p)))
              (if (same-variable? (variable high-p)
                                 (variable (cdr raised-p)))
                  (mul-poly high-p (cdr raised-p))
                  (error "Mul not in same variable, and can't change either! -- MUL-
VAR"
                         (list high-p (cdr raised-p)))))))))))

```

```

(error "Poly not in same variable, and can't change either! -- MUL-
POLY"
      (list high-p (cdr raised-p))))))))
(define (mul-terms L1 L2)
  (if (empty-termelist? L1)
      (the-empty-termelist L1)
      (add-terms (mul-term-by-all-terms (first-term L1) L2)
                 (mul-terms (rest-terms L1) L2))))
(define (mul-term-by-all-terms t1 L)
  (if (empty-termelist? L)
      (the-empty-termelist L)
      (let ((t2 (first-term L)))
        (adjoin-term
         (make-term (+ (order t1) (order t2))
                    (mul (coeff t1) (coeff t2)))
         (mul-term-by-all-terms t1 (rest-terms L))))))

(define (div-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (let ((answer (div-terms (term-list p1)
                               (term-list p2))))
        (list (tag (make-poly (variable p1) (car answer)))
              (tag (make-poly (variable p1) (cadr answer))))))
      (let ((ordered-polys (order-polys p1 p2)))
        (let ((high-p (higher-order-poly ordered-polys))
              (low-p (lower-order-poly ordered-polys)))
          (let ((raised-p (change-poly-var low-p)))
            (if (same-variable? (variable high-p)
                               (variable (cdr raised-p)))
                (div-poly high-p (cdr raised-p))
                (error "Poly not in same variable, and can't change either! -- DIV-
POLY"
                      (list high-p (cdr raised-p))))))))
(define (div-terms L1 L2)
  (define (div-help L1 L2 quotient)
    (if (empty-termelist? L1)
        (list (the-empty-termelist L1) (the-empty-termelist L1))
        (let ((t1 (first-term L1))
              (t2 (first-term L2)))
          (if (> (order t2) (order t1))
              (list (cons (type-tag L1) quotient) L1)
              (let ((new-c (div (coeff t1) (coeff t2)))
                    (new-o (- (order t1) (order t2))))
                (div-help
                 (add-terms L1
                            (mul-term-by-all-terms
                             (make-term 0 -1)
                             (mul-term-by-all-terms (make-term new-o new-c)
                                                   L2)))
                 L2
                 (append quotient (list (list new-o new-c)))))))
        (div-help L1 L2 '())))
(define (zero-pad x type)
  (if (eq? type 'sparse)
      '()
      (cond ((= x 0) '())
            ((> x 0) (cons 0 (zero-pad (- x 1) type)))
            ((< x 0) (cons 0 (zero-pad (+ x 1) type)))))

;; donno what to do when coeff `=zero?` for know just return the term-list
(define (adjoin-term term term-list)
  (define (adjoin-help term acc term-list)
    (let ((preped-term ((get 'prep-term (type-tag term-list)) term))
          (preped-first-term ((get 'prep-term (type-tag term-list))
                             (first-term term-list)))
          (empty-termelist (the-empty-termelist term-list)))
      (cond ((=zero? (coeff term)) term-list)
            ((empty-termelist? term-list) (append empty-termelist
                                                 acc
                                                 preped-term
                                                 (zero-pad (order term)
                                                       (type-tag term-list)))))

            ((> (order term) (order (first-term term-list)))
             (append (list (car term-list)) ;-> the type-tag
                     acc
                     preped-term
                     (zero-pad (- (- (order term)
                                    (order (first-term term-list)))
                                 1) (type-tag term-list)))
                     (cdr term-list)))
            ((= (order term) (order (first-term term-list)))
             (append (list (car term-list))
                     acc
                     preped-term ;-> if same order, use the new term
                     (zero-pad (- (- (order term)
                                    (order (first-term term-list)))))))))))

```

```

                (order (first-term term-list)))
                1) (type-tag term-list)
            (cddr term-list))) ;> add ditch the original term.
        (else
            (adjoin-help term
                (append acc preped-first-term)
                (rest-terms term-list))))))
        (adjoin-help term '() term-list))

(define (negate p)
    (let ((neg-p ((get 'make-polynomial (type-tag (term-list p)))
                    (variable p) (list (make-term 0 -1))))))
        (mul-poly (cdr neg-p) p))) ; cdr of neg p to eliminat the tag 'polynomial

(define (zero-poly? p)
    (define (all-zero? term-list)
        (cond ((empty-term-list? term-list) #t)
              (else
                  (and (=zero? (coeff (first-term term-list)))
                        (all-zero? (rest-terms term-list)))))))
    (all-zero? (term-list p)))

(define (equal-poly? p1 p2)
    (and (same-variable? (variable p1) (variable p2))
         (equal? (term-list p1) (term-list p2)))))

(define (the-empty-term-list term-list)
    (let ((proc (get 'the-empty-term-list (type-tag term-list))))
        (if proc
            (proc)
            (error "No proc found -- THE-EMPTY-TERMLIST" term-list))))
(define (rest-terms term-list)
    (let ((proc (get 'rest-terms (type-tag term-list))))
        (if proc
            (proc term-list)
            (error "-- REST-TERMS" term-list))))
(define (empty-term-list? term-list)
    (let ((proc (get 'empty-termlist? (type-tag term-list))))
        (if proc
            (proc term-list)
            (error "-- EMPTY-TERMLIST?" term-list))))
(define (make-term order coeff) (list order coeff))
(define (order term)
    (if (pair? term)
        (car term)
        (error "Term not pair -- ORDER" term)))
(define (coeff term)
    (if (pair? term)
        (cadr term)
        (error "Term not pair -- COEFF" term)))
;; Mixed polynomial operations. This better way to do this, was just to raise the other
types
;; to polynomial. Becuase raise works step by step, all coeffs will end up as complex
numbers.
(define (mixed-add x p) ; I should only use add-terms to do this.
    (define (zero-order L) ; And avoid all this effort. :-S
        (let ((t1 (first-term L)))
            (cond ((empty-termlist? L) #f)
                  ((= 0 (order t1)) t1)
                  (else
                      (zero-order (rest-terms L))))))
    (let ((tlst (term-list p)))
        (let ((last-term (zero-order tlst)))
            (if last-term
                (make-poly (variable p) (adjoin-term
                    (make-term 0
                        (add x (coeff last-term)))
                    tlst))
                (make-poly (variable p) (adjoin-term (make-term 0 x) tlst)))))

(define (mixed-mul x p)
    (make-poly (variable p)
        (mul-term-by-all-terms (make-term 0 x)
            (term-list p)))))

(define (mixed-div p x)
    (define (div-term-by-all-terms t1 L)
        (if (empty-termlist? L)
            (the-empty-termlist L)
            (let ((t2 (first-term L)))
                (adjoin-term
                    (make-term (- (order t1) (order t2))
                        (div (coeff t1) (coeff t2))))
                    (div-term-by-all-terms t1 (rest-terms L))))))
    (make-poly (variable p)
        (div-term-by-all-terms (make-term 0 x)
            (term-list p)))))


```

```

;; Polynomial transformation. (Operations on polys of different variables)
(define (variable-order v) ;-> var heirarchy tower. x is 1, every other
letter 0.
  (if (eq? v 'x) 1 0))
(define (order-polys p1 p2) ;-> a pair with the higher order poly `car`, and
the
  (let ((v1 (variable-order (variable p1))) ;-> lower order `cdr`
        (v2 (variable-order (variable p2))))
    (if (> v1 v2) (cons p1 p2) (cons p2 p1))))
(define (higher-order-poly ordered-polys)
  (if (pair? ordered-polys) (car ordered-polys)
      (error "ordered-polys not pair -- HIGHER-ORDER-POLY" ordered-polys)))
(define (lower-order-poly ordered-polys)
  (if (pair? ordered-polys) (cdr ordered-polys)
      (error "ordered-polys not pair -- LOWER-ORDER-POLY" ordered-polys)))

(define (change-poly-var p) ;-> All terms must be polys
  (define (helper-change term-list) ;-> change each term in term-list
    (cond ((empty-termlist? term-list) '())
          ;-> returns a list of polys with changed
var.
          (else
            (cons (change-term-var (variable p)
                           (type-tag term-list)
                           (first-term term-list))
                  (helper-change (rest-terms term-list))))))
  (define (add-poly-list acc poly-list) ;-> add a list of polys.
    (if (null? poly-list) ;-> no more polys, give me the result.
        acc
        (add-poly-list (add acc (car poly-list)) ;-> add acc'ed result to first poly
                      (cdr poly-list))) ;-> rest of the polys.
  (add-poly-list 0 (helper-change (term-list p))))
  (define (change-term-var original-var original-type term)
    (make-polynomial original-type (variable (cdr (coeff term)))) ;-> cdr eliminates
'polynomial
    (map (lambda (x)
            (list (order x) ;-> the order in x
                  (make-polynomial ;-> coeff is a poly in
                    original-type ;-> the original-type (in this example
y)
                  original-var ;-> the original-var is passed to the
coefs now
                  (list ;-> each term, is formed by
                    (list (order term) ;-> the order of the original term
                          (coeff x)))))) ;-> and the coeff of each term
in x
    (cdr (term-list (cdr (coeff term)))))) ;-> un-tagged termlist
  of
                                ;-> the coeff of the
term of y.

;; interface to rest of the system
(define (tag p) (attach-tag 'polynomial p))
(put 'add '(polynomial polynomial)
     (lambda (p1 p2) (tag (add-poly p1 p2))))
(put 'sub '(polynomial polynomial)
     (lambda (p1 p2) (tag (add-poly p1 (negate p2)))))
(put 'mul '(polynomial polynomial)
     (lambda (p1 p2) (tag (mul-poly p1 p2))))
(put 'negate '(polynomial)
     (lambda (p) (negate p)))
(put 'div '(polynomial polynomial)
     (lambda (p1 p2) (div-poly p1 p2)))
(put 'zero-poly? '(polynomial)
     (lambda (p) (zero-poly? p)))
(put 'equal-poly? '(polynomial polynomial)
     (lambda (p1 p2) (equal-poly? p1 p2)))
(put 'make 'polynomial
     (lambda (var terms) (tag (make-poly var terms)))))

;; Interface of the mixed operations.
;; Addition
(put 'add '(scheme-number polynomial) ; because it's commutative I won't define both.
Just
  (lambda (x p) (tag (mixed-add x p))) ;poly always second.
(put 'add '(rational polynomial)
     (lambda (x p) (tag (mixed-add (cons 'rational x) p)))) ;-> this is needed because
(put 'add '(real polynomial) ;-> apply-generic will
remove the
  (lambda (x p) (tag (mixed-add x p)))) ;-> tag.
(put 'add '(complex polynomial)
     (lambda (x p) (tag (mixed-add (cons 'complex x) p))))
;; Subtraction
(put 'sub '(scheme-number polynomial)
     (lambda (x p) (tag (mixed-add x (negate p)))))
(put 'sub '(polynomial scheme-number)
     (lambda (p x) (tag (mixed-add (mul -1 x) p)))))


```

```

(put 'sub '(rational polynomial)
     (lambda (x p) (tag (mixed-add (cons 'rational x) (negate p)))))

(put 'sub '(polynomial rational)
     (lambda (p x) (tag (mixed-add (mul -1 (cons 'rational x)) p)))))

(put 'sub '(real polynomial)
     (lambda (x p) (tag (mixed-add x (negate p)))))

(put 'sub '(polynomial real)
     (lambda (p x) (tag (mixed-add (mul -1 x) p)))))

(put 'sub '(complex polynomial)
     (lambda (x p) (tag (mixed-add (cons 'complex x) (negate p)))))

(put 'sub '(polynomial complex)
     (lambda (p x) (tag (mixed-add (mul -1 (cons 'complex x)) p)))))

;; Multiplication
(put 'mul '(scheme-number polynomial)
     (lambda (x p) (tag (mixed-mul x p)))))

(put 'mul '(rational polynomial)
     (lambda (x p) (tag (mixed-mul (cons 'rational x) p)))))

(put 'mul '(real polynomial)
     (lambda (x p) (tag (mixed-mul x p)))))

(put 'mul '(complex polynomial)
     (lambda (x p) (tag (mixed-mul (cons 'complex x) p)))))

;; Division
;; Using a polynomial as a divisor will leave me with negative orders. Which I don't know how
to
;; handle yet.
(put 'div '(polynomial scheme-number)
     (lambda (p x) (tag (mixed-mul (/ 1 x) p)))))

(put 'div '(scheme-number polynomial)
     (lambda (x p) (tag (mixed-div p x)))))

(put 'div '(polynomial rational) ;multiply by the denom, and divide by the numer.
     (lambda (p x) (tag (mixed-mul (make-rational (cdr x) (car x)) p)))))

(put 'div '(rational polynomial)
     (lambda (x p) (tag (mixed-div p (cons 'rational x)))))

(put 'div '(polynomial real)
     (lambda (p x) (tag (mixed-mul (/ 1.0 x) p)))))

(put 'div '(real polynomial)
     (lambda (x p) (tag (mixed-div p x)))))

(put 'div '(polynomial complex)
     (lambda (p x) (tag (mixed-mul (div 1 (cons 'complex x)) p)))))

(put 'div '(complex polynomial)
     (lambda (x p) (tag (mixed-div p (cons 'complex x)))))

'done)

(install-polynomial-package)

; this takes an extra argument type to specify if it is dense or sparse.
(define (make-polynomial type var terms)
  (let ((proc (get 'make-polynomial type)))
    (if proc
        (proc var terms)
        (error "Can't make poly of this type -- MAKE-POLYNOMIAL"
              (list type var terms)))))

; the generic negate procedure needed for subtractions.

(define (negate p)
  (apply-generic 'negate p))

; And the generic first-term procedure with its package to work with dense and
; sparse polynomials.

(define (first-term term-list)
  (let ((proc (get 'first-term (type-tag term-list))))
    (if proc
        (proc term-list)
        (error "No first-term for this list -- FIRST-TERM" term-list)))))

(define (install-polynomial-term-package)
  (define (first-term-dense term-list)
    (if (empty-termlist? term-list)
        '()
        (list
         (- (length (cdr term-list)) 1)
         (car (cdr term-list)))))

  (define (first-term-sparse term-list)
    (if (empty-termlist? term-list)
        '()
        (cadar term-list)))

  (define (prep-term-dense term)
    (if (null? term)
        '()
        (cdr term))) ;-> only the coeff for a dense term-list

  (define (prep-term-sparse term)
    (if (null? term)
        '()
        (list term))) ;-> (order coeff) for a sparse term-list

  (define (the-empty-termlist-dense) '(dense)))

```

```

(define (the-empty-term-list-sparse) '(sparse))
(define (rest-terms term-list) (cons (type-tag term-list) (cddr term-list)))
(define (empty-term-list? term-list)
  (if (pair? term-list)
      (>= 1 (length term-list))
      (error "Term-list not pair -- EMPTY-TERMLIST?" term-list)))
(define (make-polynomial-dense var terms)
  (append (list 'polynomial var 'dense) (map cadr terms)))
(define (make-polynomial-sparse var terms)
  (append (list 'polynomial var 'sparse) terms))
(put 'first-term 'sparse
  (lambda (term-list) (first-term-sparse term-list)))
(put 'first-term 'dense
  (lambda (term-list) (first-term-dense term-list)))
(put 'prep-term 'dense
  (lambda (term) (prep-term-dense term)))
(put 'prep-term 'sparse
  (lambda (term) (prep-term-sparse term)))
(put 'rest-terms 'dense
  (lambda (term-list) (rest-terms term-list)))
(put 'rest-terms 'sparse
  (lambda (term-list) (rest-terms term-list)))
(put 'empty-term-list? 'dense
  (lambda (term-list) (empty-term-list? term-list)))
(put 'empty-term-list? 'sparse
  (lambda (term-list) (empty-term-list? term-list)))
(put 'the-empty-term-list 'dense
  (lambda () (the-empty-term-list-dense)))
(put 'the-empty-term-list 'sparse
  (lambda () (the-empty-term-list-sparse)))
(put 'make-polynomial 'sparse
  (lambda (var terms) (make-polynomial-sparse var terms)))
(put 'make-polynomial 'dense
  (lambda (var terms) (make-polynomial-dense var terms)))
'done)

(install-polynomial-term-package)

```

dzy

emm, although I don't code as much as above, I think that we should't just set x's value as 1. Instead, we should sort them lexicographically, so that we can solve more complex situation, such as  $((3y+4z)x)$  add  $((3y+4z)x)$ , because the coefficient of  $x$ ,  $3y+4z$ , is also sorted so we avoid the problem when add  $(3y+4z)$  and  $(3y+4z)$

# sicp-ex-2.93

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.92) | Index | Next exercise (2.94) >>

woofy

```
(define (install-rational-package)
  ;; internal procedures
  (define (numer x) (car x))
  (define (denom x) (cdr x))
  (define (make-rat n d) (cons n d))
  (define (add-rat x y)
    (make-rat (add (mul (numer x) (denom y))
                  (mul (numer y) (denom x)))
              (mul (denom x) (denom y))))
  (define (sub-rat x y)
    (make-rat (sub (mul (numer x) (denom y))
                  (mul (numer y) (denom x)))
              (mul (denom x) (denom y))))
  (define (mul-rat x y)
    (make-rat (mul (numer x) (numer y))
              (mul (denom x) (denom y))))
  (define (div-rat x y)
    (make-rat (mul (numer x) (denom y))
              (mul (denom x) (numer y))))
  ;; interface to rest of the system
  ; same as before
)

(define p1 (make-polynomial 'x '((2 1) (0 1))))
(define p2 (make-polynomial 'x '((3 1) (0 1))))
(define rf (make-rational p2 p1))

(add rf rf)
```

# sicp-ex-2.94

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.93) | Index | Next exercise (2.95) >>

meteorgan

```
(define (greatest-common-divisor a b)
  (apply-generic 'greatest-common-divisor a b))

;; add into scheme-number package
(put 'greatest-common-divisor '(scheme-number scheme-number)
     (lambda (a b) (gcd a b)))

;; add into polynomial package
(define (remainder-terms p1 p2)
  (cadr (div-terms p1 p2)))

(define (gcd-terms a b)
  (if (empty-termlist? b)
      a
      (gcd-terms b (remainder-terms a b)))))

(define (gcd-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-poly (variable p1)
                 (gcd-terms (term-list p1)
                            (term-list p2)))
      (error "not the same variable -- GCD-POLY" (list p1 p2)))))

(put 'greatest-common-divisor '(polynomial polynomial)
     (lambda (a b) (tag (gcd-poly a b)))))
```

result of (greatest-common-divisor p1 p2): -x^2+x

Last modified : 2023-06-17 13:51:53  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-2.95

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.94) | Index | Next exercise (2.96) >>

```
(define P1 (make-polynomial 'x '((2 1) (1 -2) (0 1))))
(define P2 (make-polynomial 'x '((2 11) (0 7))))
(define P3 (make-polynomial 'x '((1 13) (0 5)))

(define Q1 (mul P1 P2))
(define Q2 (mul P1 P3))

(greatest-common-divisor Q1 Q2)
;; Value: (polynomial x (2 1458/169) (1 -2916/169) (0 1458/169))

;; So what happened? Why do we get such a strange result?

;; When gcd-terms is applied to the terms of Q1 and Q2, it returns another call
;; of gcd-terms taking the terms of Q2 and the remainder terms of Q1
;; divided by Q2 as its arguments. Unfortunately the remainder terms have
;; nonintegers as coefficients. That's why the final result has noninteger
;; coefficients.
```

<https://wqzhang.wordpress.com/2009/07/09/sicp-exercise-2-95/> (link broken)

Last modified : 2023-06-17 14:00:27  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-2.96

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.95) | Index | Next exercise (2.97) >>

meteorgan

```
;; a
(define (pseudoremainder-terms a b)
  (let* ((o1 (order (first-term a)))
         (o2 (order (first-term b)))
         (c (coeff (first-term b)))
         (divident (mul-terms (make-term 0
                                         (expt c (+ 1 (- o1 o2))))))
                  a))
    (cadr (div-terms divident b)))

(define (gcd-terms a b)
  (if (empty-termlist? b)
      a
      (gcd-terms b (pseudoremainder-terms a b)))))

;; b
(define (gcd-terms a b)
  (if (empty-termlist? b)
      (let* ((coeff-list (map cadr a))
             (gcd-coeff (apply gcd coeff-list)))
        (div-terms a (make-term 0 gcd-coeff)))
      (gcd-terms b (pseudoremainder-terms a b))))
```

Sphinxsky

```
(define (pseudo-remainder-terms L1 L2)
  (let ((f1 (first-term L1))
        (f2 (first-term L2)))
    (let ((o1 (order f1))
          (o2 (order f2))
          (c (coeff f2)))
      (let ((k (expt c (add 1 (sub o1 o2)))))
        (let ((k-terms (adjoin-term
                         (make-term 0 k)
                         (the-empty-termlist))))
          (cadr (div-terms (mul-terms k-terms L1) L2))))))

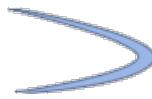
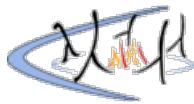
(define (gcd-coeff terms)

  (define (coeff-list terms)
    (if (empty-termlist? terms)
        '()
        (cons
          (coeff (first-term terms))
          (coeff-list (rest-terms terms)))))

  (apply gcd (coeff-list terms)))

(define (gcd-terms L1 L2)
  (if (empty-termlist? L2)
      (let ((coeff-terms (adjoin-term
                           (make-term 0 (gcd-coeff L1))
                           (the-empty-termlist))))
        (car (div-terms L1 coeff-terms)))
      (gcd-terms L2 (pseudo-remainder-terms L1 L2))))
```

# sicp-ex-2.97



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.96) | Index | Next exercise (3.1) >>

meteorgan

```
; ;a
(define (reduce-terms n d)
  (let ((gcdterms (gcd-terms n d)))
    (list (car (div-terms n gcdterms))
          (car (div-terms d gcdterms)))))

(define (reduce-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (let ((result (reduce-terms (term-list p1) (term-list p2))))
        (list (make-poly (variable p1) (car result))
              (make-poly (variable p1) (cadr result))))
      (error "not the same variable--REDUCE-POLY" (list p1 p2)))))

;;b. skip this, I had done such work many times, I'm tired of it.
```

Siki

The answer above does not follow the exercise's requirement. Before dividing n & d by gcdterms, a integerizing factor should be multiplied as stated in 2.96

master

I think that's done by gcd-terms, which presumably uses pseudoremainder-terms.

Sphinxsky

```
(define (reduce-terms n d)
  (let ((gcd-n-d (gcd-terms n d))
        (n-first (first-term n))
        (d-first (first-term d)))
    (let ((first (first-term gcd-n-d))
          (n-ord (order n-first))
          (d-ord (order d-first)))
      (let ((c (coeff first))
            (o2 (order first))
            (o1 (max n-ord d-ord)))
        (let ((k (expt c (add 1 (sub o1 o2)))))
          (let ((k-terms (adjoin-term
                           (make-term 0 k)
                           (the-empty-termlist))))
            (let ((kn (mul-terms k-terms n))
                  (kd (mul-terms k-terms d)))
              (list
                (car (div-terms kn gcd-n-d))
                (car (div-terms kd gcd-n-d)))))))))

(define (reduce-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (let ((result (reduce-terms (term-list p1) (term-list p2))))
        (list
          (make-poly (variable p1) (car result))
          (make-poly (variable p1) (cadr result))))
      (error
        "Polys not in same var -- REDUCE-POLY"
        (list p1 p2))))
```

There are problems in the test examples of this question, and the results cannot be scored. It is recommended to use the example in Exercise 2.93 for testing.

Kaihao

```
;;;
;;; a
;;;

(define (reduce-terms n d)
  (let ((gcd-L (gcd-terms n d)))
    (let ((o1 (max (order (first-term n))
                  (order (first-term d))))
          (o2 (order (first-term gcd-L)))
          (c (coeff (first-term gcd-L))))
      (let ((factor (expt c (+ 1 (- o1 o2))))))
        (let ((n1 (car (div-terms (mul-term-by-all-terms (schemenumber->term factor) n)
                                   gcd-L)))
              (d1 (car (div-terms (mul-term-by-all-terms (schemenumber->term factor) d)
                                   gcd-L))))
          (let ((gcd-coeff (apply gcd (append (termlist->coeff-list n1)
                                              (termlist->coeff-list d1)))))
            (let ((nn (car (div-terms n1 gcd-termlist)))
                  (dd (car (div-terms d1 gcd-termlist))))
              (list nn dd)))))))

(define (schemenumber->term x)
  (make-term 0 x))
(define (term->termlist term)
  (adjoin-term term (the-empty-termlist)))

(define (termlist->coeff-list term-list)
  (if (empty-termlist? term-list)
    '()
    (cons (coeff (first-term term-list))
          (termlist->coeff-list (rest-terms term-list)))))

(define (reduce-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
    (map (lambda (term-list)
           (make-poly (variable p1) term-list))
         (reduce-terms (term-list p1) (term-list p2)))
    (error "Polys not in the same var: REDUCE-POLY" (list p1 p2)))))

;;;
;;; b
;;;

(define (reduce n d)
  (apply-generic 'reduce n d))

;; add in scheme-number package
(define (reduce-integers n d)
  (let ((g (gcd n d)))
    (list (/ n g) (/ d g)))))

(put 'reduce '(scheme-number scheme-number) reduce-integers)

;; add in polynomial package
(put 'reduce '(polynomial polynomial) reduce-poly)

;; change in rational package
(define (make-rat n d)
  (let ((r (reduce n d)))
    (cons (car r)
          (cadr r)))))


```

# sicp-ex-3.1

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (2.97) | Index | Next exercise (3.2) >>

The solution here presents itself: use lexical closure.

```
(define (make-accumulator initial-value)
  (let ((sum initial-value))
    (lambda (n)
      (set! sum (+ sum n))
      sum)))
```

pluies

Lexical closure is indeed the way to go.

You can also use the trick that the initial value passed as parameter is already part of the lexical scope to save an instruction and write the accumulator as:

```
(define (make-accumulator acc)
  (lambda (x)
    (set! acc (+ x acc))
    acc))
```

Rather Iffy

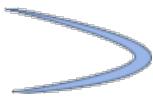
I prefer 'sum' for the result of accumulation. Acc could be confused with the accumulator object itself.

```
(define (make-accumulator sum)
  (lambda (x)
    (set! sum (+ x sum))
    sum))
```

Last modified : 2020-02-16 19:57:34  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-3.2



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.1) | Index | Next exercise (3.3) >>

Igor Saprykin

Not quite according to the task (no mf procedure), but it works.

```
(define make-monitored
  (let ((count 0))
    (lambda (f)
      (lambda (arg)
        (cond ((eq? arg 'how-many-calls?) count)
              ((eq? arg 'reset-count)
               (set! count 0)
               count)
              (else (set! count (+ count 1))
                    (f arg)))))))
```

pluies

A cleaner way, creating mf as suggested, in message-passing style:

```
(define (make-monitored function)
  (define times-called 0)
  (define (mf message)
    (cond ((eq? message 'how-many-calls?) times-called)
          ((eq? message 'reset-count) (set! times-called 0))
          (else (set! times-called (+ times-called 1))
                (function message))))
  mf)
```

mvladic

This solution allows to monitor procedures that accepts multiple arguments:

```
(define (make-monitored proc)
  (let ((count 0))
    (lambda (first . rest)
      (cond ((eq? first 'how-many-calls?) count)
            ((eq? first 'reset-count) (set! count 0))
            (else (begin (set! count (+ count 1))
                         (apply proc (cons first rest))))))))
```

Example:

```
(define m+ (make-monitored +))
(m+ 40 2)
42
(m+ 'how-many-calls?)
1
```

flamingo

Without mf procedure, using lambda:

```
(define (make-monitored f)
  (let ((count 0))
    (lambda (arg)
      (cond ((eq? arg 'how-many-calls?) count)
            ((eq? arg 'reset-count) (set! count 0))
            (else (begin (set! count (+ count 1))
                         (f arg)))))))
```

Example:

```
(define s (make-monitored sqrt))
```

```
(s 100)
10
(s 100)
10
(s 'how-many-calls?)
2
>>>
```

Daniel-Amariei

```
(define (make-accumulator acc)
  (lambda (x)
    (set! acc (+ acc x))
    acc))

(define (make-monitored f)
  (define calls (make-accumulator 0))
  (define (reset) (calls (- (calls 0))))
  (define (mf a)
    (cond ((equal? a 'how-many-calls?) (calls 0))
          ((equal? a 'reset-count) reset)
          (else (calls 1) (f a))))
  mf)

(define s (make-monitored sqrt))
(s 100) ; 10
(s 144) ; 12
(s 'how-many-calls?) ; 2
```

Shreyashm786

Here's another method , perhaps simpler

```
(define (make-monitored s)
  (define count 0)
  (define (dispatch m)
    (if (eq? m 'HowMany) count
        (begin (set! count (+ count 1)) (s m)))
  dispatch)
```

Cinderella

This solution defines the lexical scope by way of a "helper function". The anonymous function (lambda (x) ...) corresponds to the desired function "mf" and will be returned.

```
(define (make-monitored f)
  (define (mf-helper f counter)
    (lambda (x)
      (cond ((eq? x 'how-many-calls?) counter)
            ((eq? x 'reset-count) (begin (set! counter 0) counter))
            (else (begin (set! counter (+ counter 1)) (f x)))))))
  (mf-helper f 0))
```

Rather Iffy

Introduction of a counter by the let form and use of calls instead of references in the dispatch function mf.

```
(define (make-monitored f)
  (let ((counter 0))
    (define (reset-count)
      (set! counter 0))
    (define (how-many-calls?)
      counter)
    (define (mf m)
      (cond ((eq? m 'reset-count) (reset-count))
            ((eq? m 'how-many-calls?) (how-many-calls?))
            (else (begin
```

```
          (set! counter (1+ counter))
          (f m) )) )
mf))
```

Denis Manikhin

Daniel-Amariei thank you

```
(define (make-accumulator acc)
  (lambda (x)
    (set! acc (+ acc x))
    acc))

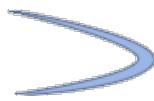
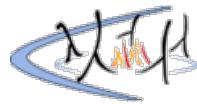
(define (make-monitored f)
  (define calls (make-accumulator 0))
  (define (mf a)
    (cond ((equal? a 'how-many-calls?) (calls 0))
            ((equal? a 'reset-count) (set! calls (make-accumulator 0)) (calls 0))
            (else (calls 1) (f a))))
  mf)
```

j-minster

Adding another to the pile.

```
(define (make-monitored f)
  (define counter 0)
  (define (call inp)
    (begin
      (set! counter (inc counter))
      (f inp)))
  (define (mf inp)
    (cond ((eq? inp 'how-many-calls?) counter)
            ((eq? inp 'reset-count) (set! counter 0))
            (else (call inp))))
  mf)
```

# sicp-ex-3.3



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.2) | Index | Next exercise (3.4) >>

Igor Saprykin

```
(define (make-account balance password)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Not enough money"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch pass m)
    (if (not (eq? pass password))
        (lambda (amount) "Wrong password")
        (cond ((eq? m 'withdraw) withdraw)
              ((eq? m 'deposit) deposit)
              (else (error "Unknown call -- MAKE-ACCOUNT"
                           m))))))
  dispatch)
```

Eivind

```
; An easier solution.

(define (make-account balance password)
  (define (withdraw amount)
    (if (>= balance amount) (begin (set! balance (- balance amount)) balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount)) balance)
  (define (dispatch p m)
    (cond ((not (eq? p password)) (lambda (x) "Incorrect password"))
          ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request -- MAKE-ACCOUNT" m))))
  dispatch)
```

altay

```
; An alternative solution with explicitly defined password validator and method
getter

(define (make-account balance initial-password)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))

  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)

  (define (get-method m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown method" m)))))

  (define (valid-password? p)
    (eq? p initial-password))

  (define (dispatch password method)
    (if (valid-password? password)
        (get-method method)
        (lambda _ "Incorrect password")))

  dispatch)
```

Carl Egbert

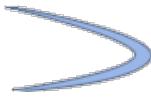
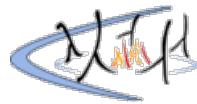
Here's a solution that uses a high-order `make-passworded` function, which could be applied to any function with an identical API to the original `make-account`:

```
;; original make-account function from the text
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount)))
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request -- MAKE-ACCOUNT"
                       m))))
  dispatch)

(define (make-passworded f)
  (define (constructor initial password)
    (define (correct-pass? attempt)
      (eq? password attempt))
    (let ((passworded-f (f initial)))
      (define (dispatch attempt x)
        (if (correct-pass? attempt)
            (passworded-f x)
            (lambda (x) "incorrect password"))))
      dispatch)))
  constructor)

(define passworded-make-account
  (make-passworded make-account))
```

# sicp-ex-3.4



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.3) | Index | Next exercise (3.5) >>

Exercise 3.4 Modify the make-account procedure of exercise 3.3 by adding another local state variable so that, if an account is accessed more than seven consecutive times with an incorrect password, it invokes the procedure call-the-cops.

Igor Saprykin

```
(define (make-account balance password)
  (define (call-the-cops) "Call the cops")
  (let ((count 0)
        (limit 7))
    (define (withdraw amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Not enough money"))
    (define (deposit amount)
      (set! balance (+ balance amount)))
    (balance)
    (define (dispatch pass m)
      (if (not (eq? pass password))
          (lambda (amount)
            (if (> count limit)
                (call-the-cops)
                (begin (set! count (+ count 1))
                       "Wrong password")))
          (begin (set! count 0)
                 (cond ((eq? m 'withdraw) withdraw)
                       ((eq? m 'deposit) deposit)
                       (else (error "Unknown call -- MAKE-ACCOUNT"
                                   m)))))))
    dispatch))
```

Kenneth

This solution has a logic error. It will actually invoke the "call-the-copes" function when there's more than 8 wrong inputs in a row, instead of 7.

Daniel-Amariei

```
(define (make-accumulator acc)
  (lambda (x)
    (set! acc (+ acc x))
    acc))

(define (make-account balance secret-password)
  (define (call-the-cops) "The cops have been called!")
  (define attempts (make-accumulator 0))
  (define (attempts-made) (attempts 0))
  (define (reset-attempts) (attempts (- (attempts 0)))))

  (define (is-correct-password? password)
    (cond ((equal? secret-password password)
           (reset-attempts) true)
          (else (attempts 1)
                false)))

  (define (withdraw amount)
    (cond ((>= balance amount)
           (set! balance (- balance amount))
           balance)
          (else
```

```

        "Insufficient funds")))

(define (deposit amount)
  (set! balance (+ balance amount))
  balance)

(define (dispatch password m)
  (cond ((not (is-correct-password? password))
         (if (> (attempts-made) 7)
             (lambda (x) (call-the-cops))
             (lambda (x) "Incorrect password")))
        ((equal? m 'withdraw) withdraw)
        ((equal? m 'deposit) deposit)
        (else (error "Unknown request: MAKE-ACCOUNT" m)))
  dispatch)

```

eliyak

I put all the password-checking functionality into a separate internal function called `correct-password?`. Unfortunately my implementation of scheme does not know how to `(call-the-cops)`.

```

(define (make-account balance password)
  (define bad-password-count 0)
  (define (correct-password? p)
    (if (eq? p password)
        (set! bad-password-count 0)
        (begin
          (if (= bad-password-count 7)
              (call-the-cops)
              (set! bad-password-count (+ bad-password-count 1)))
          (display "Incorrect password")
          #f)))
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch p m)
    (if (correct-password? p)
        (cond
          ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request -- MAKE-ACCOUNT" m)))
        (lambda (x) (display ""))))
  dispatch)

```

anonymous

(this code was contributed at the 27 Dec 2013 edit)

```

(define (make-account balance password)
  (let ((bad-passwords 0))
    (define (withdraw amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          (print "Insufficient funds")))
    (define (deposit amount)
      (set! balance (+ balance amount))
      balance)
    (define (dispatch p m)
      (if (good-password? p)
          (cond ((eq? m 'withdraw) withdraw)
                ((eq? m 'deposit) deposit)
                (else (error "Unknown request -- MAKE-ACCOUNT" m)))
          (lambda (x) (print "Incorrect password") (newline))))
    (define (good-password? p)
      (cond ((eq? p password)
             (set! bad-passwords 0)
             true)
            ((< bad-passwords 7)
             (set! bad-passwords (+ bad-passwords 1))
             false)
            (else
              (call-the-cops))))
    (define (call-the-cops)
      (print "Cops called!"))

```

```
false)
(dispatch))
```

Caleb Gossler

I wrapped `dispatch` in a function that handles the password checking. It keeps `dispatch` much cleaner.

```
(define (make-account password balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount)))
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request -- MAKE-ACCOUNT"
                      m))))
  (password-protect password dispatch))

(define (password-protect user-password f)
  (define invalid-attempts 0)
  (lambda (password m)
    (cond [= invalid-attempts 7]
          (call-the-cops)
          [(eq? password user-password)
           (set! invalid-attempts 0)
           (f m)]
          [else
           (set! invalid-attempts (+ 1 invalid-attempts))
           (error "Incorrect password")])))

(define (call-the-cops)
  (error "The cops have been called!"))
```

Han Chan

My implementation is much simpler. I just wrap the `dispatch` function with an "auth-layer" function, which will do the authentication. Then I wrap the "auth-layer" function with a "secure-layer", which will check the error counter and decide to call the cops or not.

```
(define (secure-make-account balance password)

  (define error-count 0)

  (define (call-the-police) "Too much errors, we have called LAPD")

  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount)) balance)
        "Insufficient funds"))

  (define (deposit amount)
    (set! balance (+ balance amount)) balance)

  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request -- MAKE-ACCOUNT" m))))
  )

  (define (auth-layer pass m)
    (if (eq? pass password)
        (dispatch m)
        (lambda (x) (begin (set! error-count (+ 1 error-count))
                           "Incorrect password"))))

  (define (secure-layer pass m)
    (if (= 2 error-count)
        (lambda (x) (call-the-police))
        (auth-layer pass m)))

  secure-layer)
```

roy-tobin

The spec says "accessed more than seven consecutive times..." And so the above code has two problems: first,  $2 \neq 7$  and second, "consecutive" means "following one another without interruption." Thus, a matching (i.e. successful) password must reset the error-count.

Sphinxsky

```
(load "make-monitored.scm")

(define (make-account balance password)

  (define (withdraw amount)
    (if (>= balance amount)
        (begin
          (set! balance (- balance amount))
          balance)
        "insufficient funds"))

  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)

  (define (call-the-cops) "We have already called the police for your actions.")

  (define (password-errors-logic monitored)
    (lambda (m)
      (let ((wrong-times (monitored 'how-many-calls?)))
        (if (= wrong-times 6)
            (begin
              (monitored 'reset-count)
              (call-the-cops))
            (monitored
              (if (eq? m 'reset-count)
                  'reset-count
                  (- 6 wrong-times)))))

  (define counter
    (password-errors-logic
      (make-monitored
        (lambda (m)
          (string-append
            "Incorrect password: "
            "You can still enter "
            (number->string m)
            " wrong times.")))))

  (define (dispatch p m)
    (if (eq? p password)
        (begin
          (counter 'reset-count)
          (cond ((eq? m 'withdraw) withdraw)
                ((eq? m 'deposit) deposit)
                (else (error "unknown request -- MAKE-ACCOUNT" m))))
        counter))

  dispatch)
```

zxyMike93

My solution is not so original but it's plain and simple.

```
(define (make-account balance password)

  (define psswd password)

  (define lock 0)

  (define (count-incorrect-password amount)
    (begin (set! lock (+ lock 1))
           (display "Incorrect password")
           (newline)))

  (define (call-the-cops amount)
    (error "Bee-boo Bee-boo Bee-boo"))

  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance
```

```

        (- balance amount))
balance)
"Insufficient funds"))
(define (deposit amount)
  (set! balance (+ balance amount))
balance)

(define (dispatch pw m)
  (cond [(eq? pw psswd)
         (cond ((eq? m 'withdraw) withdraw)
               ((eq? m 'deposit) deposit)
               (else (error "Unknown request:
MAKE-ACCOUNT" m)))]
        [else
         (cond [(>= lock 6) call-the-cops]
               [else count-incorrect-password]))])

dispatch)

```

Carl Egbert

A solution that uses a high-order, reusable `make-passworded` function, which should work with any function with an identical API to `make-account`, and also reuses the `make-monitored` solution from exercise 3.2:

```

;; original function from the text
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount)))
  balance)
(define (dispatch m)
  (cond ((eq? m 'withdraw) withdraw)
        ((eq? m 'deposit) deposit)
        (else (error "Unknown request -- MAKE-ACCOUNT"
                     m))))
dispatch)

;; solution from exercise 3.2
(define (make-monitored f)
  (let ((count 0))
    (define (how-many-calls?)
      count)
    (define (reset-count)
      (set! count 0))
    (define (monitored x)
      (begin
        (set! count (+ 1 count))
        (f x)))
    (define (dispatch x)
      (cond
        ((eq? x 'how-many-calls?) (how-many-calls?))
        ((eq? x 'reset-count) (reset-count))
        (else (monitored x))))
    dispatch))

(define (make-passworded f)
  (define (failed-entry x) "incorrect password")
  (define (call-the-cops x) "weeeeeoooo weeeeeoooo weeeeeoooo"))

  (define (constructor initial password)
    (define (correct-pass? attempt)
      (eq? password attempt))
    (define try-pass?
      (make-monitored correct-pass?))
    (define (reset-login-attempts)
      (try-pass? 'reset-count))
    (define (failed-attempts)
      (try-pass? 'how-many-calls?))

    (let ((passworded-f (f initial)))
      (define (dispatch attempt x)
        (if (try-pass? attempt)
            (begin (reset-login-attempts)
                   (passworded-f x))
            (if (< 6 (failed-attempts))
                call-the-cops
                failed-entry)))))

```

```

        dispatch))
constructor)

(define passworded-make-account
  (make-passworded make-account))

```

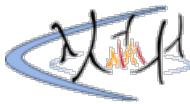
roy-tobin

Unique here is the straightforward logic cascade at the cond expression that handles all five possible contingencies in as many lines of code.

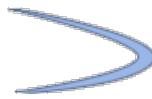
```

(define (make-account balance passwd)
  (let ((access-failures 0))
    (define (call-the-cops) (error "Out of the car, longhair!"))
    (define (bad-passwd dontcare) "Incorrect password")
    (define (withdraw amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount)) balance)
          "Insufficient funds"))
    (define (deposit amount)
      (set! balance (+ balance amount)))
    (balance)
    (define (dispatch phrase m)
      (if (eq? phrase passwd)
          (set! access-failures 0)
          (set! access-failures (+ 1 access-failures)))
      (cond ((> access-failures 7) (call-the-cops))
            ((> access-failures 0) bad-passwd)
            ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit) deposit)
            (else (error "unknown request -- MAKE-ACCOUNT" m))))
    dispatch)))

```



# sicp-ex-3.5



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.4) | Index | Next exercise (3.6) >>

x3v

Using the monte-carlo procedure as defined in the book, and the exact->inexact primitive procedure to turn a fraction into a float:

```
(define (predicate x1 x2 y1 y2 radius)
  (<= (+ (square (random-in-range x1 x2))
           (square (random-in-range y1 y2)))
       radius))

(define (estimate-integral P trials x1 x2 y1 y2)
  (* (monte-carlo trials (lambda () (P x1 x2 y1 y2 1)))
     4))
;; whereby 4 is the area of the rectangle containing the unit circle (@ * 2)

(exact->inexact (estimate-integral predicate 100000 -1.0 1.0 -1.0 1.0)) ;~3.141
```

Given:

```
(define (monte-carlo trials experiment)
  (define (iter trials-remaining trials-passed)
    (cond ((= trials-remaining 0)
           (/ trials-passed trials))
          ((experiment)
           (iter (- trials-remaining 1) (+ trials-passed 1)))
          (else
           (iter (- trials-remaining 1) trials-passed))))
  (iter trials 0))
(define (random-in-range low high)
  (let ((range (- high low)))
    (+ low (random range))))
```

The solution is:

```
(define (P x y)
  (< (+ (expt (- x 5) 2)
            (expt (- y 7) 2))
      (expt 3 2)))
(define (estimate-integral P x1 x2 y1 y2 trials)
  (define (experiment)
    (P (random-in-range x1 x2)
       (random-in-range y1 y2)))
  (monte-carlo trials experiment))
```

Test:

```
(estimate-integral P 2.0 8.0 4.0 10.0 100)
```

Then we can estimate pi with the fact that a circle area is  $(\pi * r^2)$ .

Hence  $\pi \approx (\text{Monte Carlo results} * \text{rectangle area}) / r^2$

```
(define pi-approx
  (/ (* (estimate-integral P 2.0 8.0 4.0 10.0 10000) 36)
      9.0))
pi-approx
```

Which gave 3.1336 during my test.

This function has to be tested under MIT Scheme, neither gambit-scheme or SISC implements (random) - actually (random) is not part of R5RS nor SRFI.

NB: using 2.0 instead of 2 in estimate-integral is primordial. If you pass two integers to (random-in-range low high), it will return another integer strictly inferior to your 'high' value — and this completely screws the Monte-Carlo method (it then estimates pi to ~3.00).

Using Racket's built-in (random), which returns a float between 0 and 1, the following set of functions can be used to find the numerical integral of a function with a predicate, using the provided (monte-carlo) function:

```
(define (random-in-range low high)
  (let ([range (- high low)])
    (+ low (* (random) range))))

(define (get-area lower-x lower-y upper-x upper-y)
  (* (- upper-x lower-x)
     (- upper-y lower-y)))

(define (estimate-integral pred lower-x lower-y upper-x upper-y trials)
  (let ([area (get-area lower-x lower-y upper-x upper-y)]
        [experiment (lambda () (pred (random-in-range lower-x upper-x)
                                     (random-in-range lower-y upper-y)))]
        (* area (monte-carlo trials experiment))))
```

Testing with a large number of trials:

```
(estimate-integral (lambda (x y) (<= (+ (* x x) (* y y)) 1.0)) -1.0 -1.0 1.0 1.0 100000000)
3.14130636
```

Shawn I think the procedure estimate-integral is not very nicely formulated in the text above, in my opinion, a better solution for this exercise would be:

```
(define (estimate-integral P x1 x2 y1 y2 trials)
  (* (* (- x2 x1)
         (- y2 y1))
     (monte-carlo trials P)))

(define (in-circle)
  (>= 1 (+ (square (random-in-range -1.0 1.0))
            (square (random-in-range -1.0 1.0)))))

(define (random-in-range low high)
  (let ([range (- high low)])
    (+ low (random range)))

(define (estimate-pi)
  (estimate-integral in-circle -1.0 1.0 -1.0 1.0 1000))

;; monte carlo procedure
(define (monte-carlo trials experiment)
  (define (iter trials-remaining trials-passed)
    (cond ((= trials-remaining 0)
           (/ trials-passed trials))
          ((experiment)
           (iter (- trials-remaining 1) (+ trials-passed 1)))
          (else
           (iter (- trials-remaining 1) trials-passed))))
  (iter trials 0))
```

We should leave the job of generating random numbers to the predicate, instead of intertwining it with estimate-integral. If we do it like this, we cannot adapt estimate-integral to other predicates.

athird If you're using Racket the random function doesn't work as described in the text. You might want to use this instead:

```
(define (random-in-range low high)
  (let ([range (- high low)])
    (+ low (* (random) range))))
```

gambiteer

Gambit has (random-real) for a random inexact real between 0 and 1 and (random-integer n) for a random integer between 0 and n.

martin256

I don't like Shawn solution, I think a predicate should not concern about generating random numbers, that is the job of the experiment. The circle predicate should check whether a point is inside a circle or not, the experiment determines if a random point is inside a predicate, this way you can reuse other predicates.

Dewey

To martin256. If you look closely, you'll know in Shawn solution, the circle predicate IS the

experiment. Just same with the `cesaro-test` in the book.

```
(define (cesaro-test)
  (= (gcd (rand) (rand)) 1))
```

GP

Abstraction layers can be used here to build more modulized solution making it more suitable to extend to larger system as we have learnt from Chap. 2 Data Abstraction.

2 points to be notified

1. For estimate-integral, instead of hardcode boundaries in test function P, it is better to define P as function taking boundaries as input. Since boundaries define the working region/area to calculate the integral, the concept of working region/area is generic for all kinds of tests in terms of integral estimation. Therefore, it is better to take directly the square/area/region object as parameter instead of x y coordinates. With the objects methods to calculate area and selectors to get boundary points, it is possible to manipulate things at a higher level and thus more intuitive and modulized.
2. For p-estimate, user only needs to input number of trials, as this is the only thing that change the accuracy of the result. Better to hide other paramters by creating abstraction layers with other procedures as we learnt from Chap. 1 Procedural Abstraction.

```
;; generator and selector for point
(define (make-point x y)
  (cons x y))

(define (xp point)
  (car point))

(define (yp point)
  (cdr point))

(define (mid-point p1 p2)
  (let ((x1 (xp p1))
        (x2 (xp p2))
        (y1 (yp p1))
        (y2 (yp p2)))
    (define new-x (/ (+ x2 x1) 2))
    (define new-y (/ (+ y2 y1) 2))
    (make-point new-x new-y)
  ))

;; generator and selector for circle
(define (make-circle pc r)
  (list pc r))

(define (get-c circ)
  (car circ))

(define (get-r circ)
  (cadr circ))

;; generator and selector for square
(define (make-square p1 p2)
  (list p1 p2))

(define (p1st-square square)
  (car square))

(define (p2nd-square square)
  (cadr square))

(define (area-square square)
  (let ((p1 (p1st-square square))
        (p2 (p2nd-square square)))
    (define x1 (xp p1))
    (define x2 (xp p2))
    (define y1 (yp p1))
    (define y2 (yp p2))
    (abs (* (- x2 x1) (- y2 y1)))
  ))

;; procedure to pick one random point inside square
(define rand-in-square
  (lambda (square)
    (let ((x1 (xp (p1st-square square)))
          (x2 (xp (p2nd-square square)))
          (y1 (yp (p1st-square square)))
          (y2 (yp (p2nd-square square))))
      (make-point (random-in-range x1 x2)
                  (random-in-range y1 y2))))
```

```

;; estimate-integral taking square as input instead of x y coordinates
(define (estimate-integral P square trials)
  (monte-carlo trials (P square)))

;; define unit circle and square used for estimation
(define p01 (make-point -1. -1.))
(define p02 (make-point 1. 1.))
(define pm (mid-point p01 p02))
(define unit-square (make-square p01 p02))
(define unit-cir (make-circle pm 1.))

; helper function to calculate square
(define (sqr x)
  (* x x))

;; generic in-circle test for point in any square and any circle
(define (in-circle? square cir)
  (lambda ()
    (let ((xc (xp (get-c cir)))
          (yc (yp (get-c cir)))
          (r (get-r cir)))
        (p_rand (rand-in-square square)))
      (define x_rand (xp p_rand))
      (define y_rand (yp p_rand))
      (<= (+ (sqr (- x_rand xc)) (sqr (- y_rand yc))) (sqr r)))))

;; specific test function with unit circle for pi estimate
(define in-unit-cir?
  (lambda (square)
    (in-circle? square unit-cir)))

;; p-estimate: user only needs to input number of trials
(define (p-estimate trials)
  (* (area-square unit-square)
     (estimate-integral in-unit-cir? unit-square trials)))

```

Test:

```
(p-estimate 1000000) ;;-> 3.141292
```

---

Rob

If you just want to estimate PI, the actual values do not matter. Just take a radius and try to estimate. Actually, which radius you take does also not matter that much (when working with floats)

```

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

; (r * 2)^2 * integral = r^2 * PI
; (2)^2 * integral = PI
; PI = 4 * integral
(define (calculatePI integral)
  (* integral 4.0))

(define (monte-carlo trials experiment)
  (define (iter trails-remaining trials-passed)
    (cond ((= trails-remaining 0)
           (/ trials-passed trials))
          ((experiment)
           (iter (- trails-remaining 1) (+ trials-passed 1)))
          (else
           (iter (- trails-remaining 1) trials-passed))))
  (iter trials 0))

(define (estimate-integral radius trials)
  (define squareRadius (square radius))
  (define diameter (* 2.0 radius))
  (define (predicate)
    (let ((randomX (random diameter))
          (randomY (random diameter)))
      (<= (+ (square (- randomX radius)) (square (- randomY radius))) squareRadius)))
  (monte-carlo trials predicate))

(define (estimate-pi radius trials)
  (calculatePI (estimate-integral radius trials)))

(estimate-pi 1.0 100000); 3.13848

```

joshroybal

I tried to make it a little more general.  
test runs  
(estimate-integral in-circle? 2.0 8.0 4.0 10.0 1000)  
=> 3.24  
(estimate-integral in-circle? -1.0 1.0 -1.0 1.0 1000)  
=> 28.44

```
(define (in-circle? x1 x2 y1 y2)
  (let ((center (cons (/ (+ x1 x2) 2.0) (/ (+ y1 y2) 2.0)))
        (r (/ (min (- x2 x1) (- y2 y1)) 2.0))
        (x (random-in-range x1 x2))
        (y (random-in-range y1 y2)))
    (<= (+ (square (- x (car center)))
            (square (- y (cdr center))))
         (square r)))))

(define (estimate-integral predicate x1 x2 y1 y2 trials)
  (let ((area (* (- x2 x1) (- y2 y1))))
    (* area (monte-carlo trials (lambda () (predicate x1 x2 y1 y2))))))
```

j-minster

A specialised solution, not general. Using the random function from the sicp lang in Racket.  
Just doing 10000 simulations, since this wasn't specified in the problem.

```
#lang sicp

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

(define (in-unit-circ? x y)
  (<= (+ (square x)
          (square y))
      1))

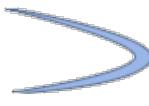
(define (rand-between x1 x2)
  (+ x1 (random (- x2 x1)))))

(define (estimate-integral P x1 x2 y1 y2) ; x1 < x2 && y1 < y2
  (define area (abs (* (- x2 x1) (- y2 y1))))
  (* area
     (monte-carlo 10000 (lambda () (P (rand-between x1 x2) (rand-between y1 y2))))))

;; (estimate-integral in-unit-circ? -1.0 1.0 -1.0 1.0)
;; => 3.148
```



# sicp-ex-3.6



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.5) | Index | Next exercise (3.7) >>

```
(define rand
  (let ((x random-init))
    (define (dispatch message)
      (cond ((eq? message 'generate)
             (begin (set! x (rand-update x))
                   x))
            ((eq? message 'reset)
             (lambda (new-value) (set! x new-value))))
        dispatch)))
```

Test:

```
(define random-init 0)
(define (rand-update x) (+ x 1)) ; A not-very-evolved PRNG
(rand 'generate)
; 1
(rand 'generate)
; 2
((rand 'reset) 0)
; 0
(rand 'generate)
; 1
```

It's interesting to notice that the lambda returned by a call to (rand 'reset) still has the closure we created as lexical scope:

```
x
; Error: undefined variable 'x'.
```

hi-artem

Here, inside (cond ..) construct, using "begin" is unnecessary.

Shawn

We could also define rand as a procedure instead of a variable:

```
(define (rand arg)
  (let ((x random-init))
    (cond ((eq? arg 'generate)
           (set! x (rand-update x))
           x)
          ((eq? arg 'reset)
           (lambda (new-value) (set! x new-value))))))
```

pa3

Shawn's version will not work actually. It will always return the result of evaluating (rand-update random-input), because the state (x variable) is being recreated and reset to random-input each time rand function invoked.

tf3

Just chiming in with pa3, Shawn is really wrong. Even I made the same mistake and thankfully came here to compare. The block (let ((x random-init)) ..) is part of the function's body, so it would be called every time the function is called, which would defeat the purpose of assignment. The book didn't highlight this, or maybe we didn't read too closely, but using the variable approach would prevent the variable assignment from executing except the first time.

joew

```
#lang sicp
;"random" number generator
(define (rand-update x)
  (let ((a 27) (b 26) (m 127))
```

```

(modulo (+ (* a x) b) m))

(define x1 (rand-update 0))
(define x2 (rand-update x1))
(define x3 (rand-update x2))
(define x4 (rand-update x3))
(define x5 (rand-update x4))
x1
x2
x3
x4
x5
;function that returns a closure
;closures seem to be the way to initialize objects with state in scheme
(define (r)
  (let ((seed 0))
    (define (dispatch m)
      (cond ((eq? m 'reset) (lambda (x)(set! seed x)))
            ((eq? m 'generate) (begin (set! seed (rand-update seed))
                                         seed))
            (else error "invalid operation")))
    dispatch))
;define rand in terms of r and the closure it returns
(define rand (r))
;test that rand works first with no args assuming 0 as seed
(rand 'generate)
(rand 'generate)
(rand 'generate)
;test that pattern reoccurs
((rand 'reset) 0)
(rand 'generate)
(rand 'generate)
(rand 'generate)
;test the rand works using a new seed
((rand 'reset) 42)
(rand 'generate)
(rand 'generate)
(rand 'generate)
;test the pattern reoccurs
((rand 'reset) 42)
(rand 'generate)
(rand 'generate)
(rand 'generate)

```

codybartfast

Just for the laughs here are two (functionally) identical implementations, one using define to create named procedures:

```

(define (make-rand)
  (define x 0)
  (define (set-x! new-x)
    (set! x new-x))
  (define (dispatch message)
    (cond
      ((eq? message 'generate)
       (set! x (rand-update x))
       x)
      ((eq? message 'reset)
       set-x!)
      (else ((error "Unknown Message - " message)))))

  dispatch)

```

and the other using lambda to create anonymous procedures:

```

(define (make-rand)
  (let ((x 0))
    (lambda (message)
      (cond
        ((eq? message 'generate)
         (set! x (rand-update x))
         x)
        ((eq? message 'reset)
         (lambda (new-x) (set! x new-x)))
        (else ((error "Unknown Message - " message)))))))

```

Usage:

```

(define (make-rand)
  ...)
(define my-rand (make-rand))
((my-rand 'reset) 42)

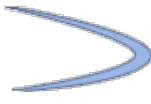
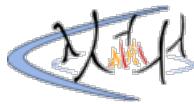
```

```
| (my-rand 'generate)
| (my-rand 'generate)
```

---

Last modified : 2021-05-06 15:14:00  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-3.7



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.6) | Index | Next exercise (3.8) >>

x3v

Added a separate auth layer in the make-account procedure. Enables message-passing style implementation for "chained" joint accounts.

```
(define (make-account balance password)
  (define incorrect-count 0)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount)) balance)
        "insufficient"))
  (define (deposit amount)
    (set! balance (+ balance amount)) balance)
  (define (issue-warning)
    (if (> incorrect-count 7)
        (error "the cops are on their way")
        (error (- 7 incorrect-count) 'more 'attempts)))
  (define (auth-layer pw . m)
    (cond ((null? m) (eq? pw password))
          ((eq? pw password) (dispatch (car m)))
          (else (begin (set! incorrect-count (+ incorrect-count 1))
                        (issue-warning)))))

  (define (dispatch m)
    (set! incorrect-count 0)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request" m))))
  auth-layer)

(define (make-joint acc pw-prev pw-next)
  (define (dispatch pw . m)
    (if (null? m)
        (eq? pw pw-next)
        (acc (if (eq? pw pw-next) pw-prev pw-next) (car m))))
  (if (acc pw-prev)
      dispatch
      (error "Incorrect password to original account" pw-prev)))

(define peter-acc (make-account 100 'open-sesame))
(define paul-acc (make-joint peter-acc 'open-sesame 'rosebud))
(define pan-acc (make-joint paul-acc 'rosebud 'vvv))

;; tests
((pan-acc 'vvv 'deposit) 100) ; 200
((peter-acc 'open-sesame 'deposit) 100) ; 300
((paul-acc 'rosebud 'deposit) 100) ; 400
((peter-acc 'rosebud 'deposit) 100) ; error as intended
```

woofy

arbitrary number of layers of joint account:

```
(define (make-account balance password)

  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount)) balance)
        "Insufficient funds"))

  (define (deposit amount)
    (set! balance (+ balance amount)) balance)

  (define (dispatch pwd m)
    (if (eq? m 'auth)
        (lambda () (eq? pwd password))
        (if (eq? pwd password)
```

```

(cond ((eq? m 'withdraw) withdraw)
      ((eq? m 'deposit) deposit)
      (else (error "Unknown request -- MAKE-ACCOUNT" m)))
(error "Incorrect password")))

(dispatch)

(define (make-joint account orig-pwd joint-pwd)

  (define (withdraw amount) ((account orig-pwd 'withdraw) amount))
  (define (deposit amount) ((account orig-pwd 'deposit) amount))

  (define (dispatch pwd m)
    (if (eq? m 'auth)
        (lambda () (eq? pwd joint-pwd))
        (if (eq? pwd joint-pwd)
            (cond ((eq? m 'withdraw) withdraw)
                  ((eq? m 'deposit) deposit)
                  (else (error "Unknown request -- MAKE-ACCOUNT" m)))
            (error "Incorrect password")))

    (if ((account orig-pwd 'auth))
        dispatch
        (error "Incorrect original password of target account")))

; test
(define peter-acc (make-account 100 'open-sesame))
(define paul-acc (make-joint peter-acc 'open-sesame 'rosebud)) ;joint account
(define woofy-acc (make-joint paul-acc 'rosebud 'longleg)) ; joint joint account

((peter-acc 'open-sesame 'withdraw) 50) ;50
((paul-acc 'rosebud 'withdraw) 10) ;40
((peter-acc 'open-sesame 'deposit) 0) ;40
((woofy-acc 'longleg 'withdraw) 33) ;7
((peter-acc 'open-sesame 'deposit) 0) ;7

((woofy-acc 'rosebud 'withdraw) 100) ;wrong pwd
((woofy-acc 'open-sesame 'withdraw) 100) ;wrong pwd

```

See [sicp-ex-3.3-3.4](#) for the definition of **make-account**.

```

(define (make-joint account old-pass new-pass)
  (and (number? ((account old-pass 'withdraw) 0))
       (lambda (pass msg)
         (if (eq? pass new-pass)
             (account old-pass msg)
             (account 'bad-pass 'foo)))))) ;increment bad-passwords

```

Alternative implementation:

```

(define (call-the-cops)
  (display "calling the cops\n"))

(define (password-protect password subject)
  (let ((num-attempts 0))
    (lambda (provided-password msg)
      (if (eq? provided-password password)
          (begin (set! num-attempts 0)
                 (subject msg))
          (begin (set! num-attempts (+ 1 num-attempts))
                 (when (>= num-attempts 7)
                   (call-the-cops))
                 (lambda (arg . rest) "invalid password")))))

(define (make-account password balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch msg)
    (cond ((eq? msg 'withdraw) withdraw)
          ((eq? msg 'deposit) deposit)
          (else (error "Unknown request -- MAKE-ACCOUNT"
                      msg))))
  (password-protect password dispatch))

(define (make-joint-account account original-password password)
  (define (dispatch msg)

```

```
(account original-password msg)
(password-protect password dispatch))
```

```
(define (make-account balance secret-words)

  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount)) balance)
        "Insufficient funds"))

  (define (deposit amount)
    (set! balance (+ balance amount)) balance)

  (define (make-join secret-words) (set-sw secret-words))

  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          ((eq? m 'make-join) make-join)
          (else (error "Unknown request -- MAKE-ACCOUNT"
                      m)))))

  (define (set-sw secret-words)
    (lambda (pw m)
      (cond ((eq? pw secret-words) (dispatch m))
            (else (error "Permission Denied -- MAKE-ACCOUNT")))))

  (set-sw secret-words)))
```

CrazyAlvaro

Implements above are correct, just provide another way to do it

```
(define (make-account balance pass-origin)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))

  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)

  (define (make-joint another-pass)
    (dispatch another-pass))

  (define (dispatch password)
    (lambda (pass m)
      (if (eq? pass password)
          (cond ((eq? m 'withdraw) withdraw)
                ((eq? m 'deposit) deposit)
                ((eq? m 'make-joint) make-joint)
                (else (error "Unknown request -- MAKE-ACCOUNT"
                            m)))
          (lambda (x)
            "Incorrect password"))))

  (dispatch pass-origin)))
```

drugkeeper

This checks if you keyed in the 1st-password correctly when making the joint acc, by trying out the 1st pass to deposit 0 dollars into the account.

When depositing or withdrawing, it takes your password and checks it against new-pass, if its correct then it converts the password to 1st-pass and passes the message to the original account.

You can also make many joint accounts all sharing together as all the passwords are converted to the 1st-pass.

```
(define (make-joint acc 1st-pass new-pass)
  (define (dispatch p m)
    (cond ((eq? p new-pass) (acc 1st-pass m))
          (else (error "Wrong password"))))
    (begin ((acc 1st-pass 'deposit) 0) ;this does a clever check on whether 1st-pass is
           correct.
           dispatch)))
```

master

This solution uses a password list to keep track of all the authorized passwords. Anybody

can use any of the passwords contained in the list, however. I don't really see any way around that because there aren't any usernames, and there's no easy way to tell at the moment under what name the account is invoked. No need to make a dummy transaction as the only way to create a joint account is via an internal procedure in `make-account`.

```
(define (make-account balance password)
  (let ((tries 0)
        (password-list (list password)))
    (define (withdraw amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Insufficient funds"))
    (define (deposit amount)
      (set! balance (+ balance amount))
      balance)
    (define (contains? p plist)
      (if (not (eq? (memq p plist) #f))
          #t
          (lambda (x) "Incorrect password")))
    (define (add-user pass)
      (set! password-list (cons pass password-list)))
    (define (dispatch p m)
      (if (contains? p password-list)
          (cond ((eq? m 'withdraw) withdraw)
                ((eq? m 'deposit) deposit)
                ((eq? m 'add-user) add-user)
                (else (error "Unknown request: MAKE-ACCOUNT"
                             m)))
          (lambda (x) "Incorrect password")))
    dispatch))

(define (make-joint account password new-password)
  ((account password 'add-user) new-password)
  account)
```

santi

```
#!/usr/bin/racket
#lang racket

(define (make-account balance pass)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (incorrect-pass _)
    "Incorrect password")

  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          ((eq? m 'joint) secure-dispatch)
          (else (error "Unknown request: MAKE-ACCOUNT"
                       m)))))

  (define (secure-dispatch new-pass)
    (lambda (input-pass m)
      (if (eq? input-pass new-pass)
          (dispatch m)
          incorrect-pass
        )))

  (secure-dispatch pass))

; (define acc (make-account 100))

(define (make-joint acc pass new-pass)
  ((acc pass 'joint) new-pass)
  )

;tests

(define peter-acc (make-account 100 'open-sesame))
(define paul-acc (make-joint peter-acc 'open-sesame 'rosebud))

((peter-acc 'open-sesame 'withdraw) 10) ; 90
```

```
((paul-acc 'rosebud 'withdraw) 10) ; 80
((paul-acc 'open-sesame 'withdraw) 10) ; Incorrect password
```

denis manikhin

thank you very much x3v for idee with auth-layer

```
#lang sicp

(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request -- MAKE-ACCOUNT"
                       m)))))

(define (make-passworded f f-password)
  (lambda (password . z)
  (cond
    ((null? z) (eq? password f-password))
    ((eq? password f-password) (f (car z)))
    (else (lambda _ "incorrect-password")))))

(define (make-passworded-acc balance password)
  (make-passworded (make-account balance) password))

(define acc-denis ( make-passworded-acc 1000 'denis-secret))

((acc-denis 'denis-secret 'withdraw) 20)
((acc-denis 'denis-secret 'withdraw) 20)

(define (make-joint main-acc main-password joint-password)
  (if (main-acc main-password)
      (make-passworded (lambda (x) (main-acc main-password x)) joint-password)
      (display "incorrect password for main account ")))

(define acc-aisha (make-joint acc-denis 'denis-secret 'aisha-secret))

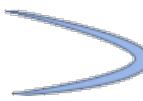
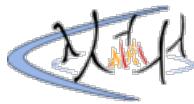
((acc-aisha 'aisha-secret 'deposit) 500)

(define acc-victoria (make-joint acc-aisha 'aisha-secret 'victoria-secret))

((acc-victoria 'victoria-secret 'deposit) 400)

(define acc-oxana (make-joint acc-denis 'oxana-secret 'oxana-secret))
```

# sicp-ex-3.8



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.7) | Index | Next exercise (3.9) >>

bro\_chenzox

I have tried to use all of solutions but no any of them is accepting my test that you can find below the solution. I just have a little bit redone aaa's answer.

```
(define f
  (let ((init 0))
    (lambda (x)
      (set! init (- x init))
      (- x init))))
```

```
(+ (f 0) (f 1))
(+ (f 1) (f 0))
(+ (f 0) (f 1))
(+ (f 1) (f 0))
(+ (f 1) (f 0))
(+ (f 0) (f 1))
(+ (f 0) (f 1))
```

chm

Lot's of solutions already, but I figured I might add this since it's pretty simple and in my opinion more elegant than some of the others, since it's a function that could be useful in some situations other than just solving this exercise

```
(define f
  (let ((count 1))
    (lambda (x)
      (set! count (* count x))
      count)))
```

master

Nice idea but unfortunately it isn't quite right, the answer depends on whether `f` has ever been called with an argument of zero, whereas I think it is supposed to depend only on the evaluation order.

tf3

There's no problem in his code, as it satisfies the requirements of the question.

But to be frank, here we have two calls to the same function `f` and we are summing them like `f(0) + f(1)`. Ordinarily, if you see a function `f` which returns numerical values (as is the case here) which are then added in successive calls to the function (eg. like during recursive calls to fibonacci), you wouldn't want them to behave in such a way in the first place. I mean such a behaviour is definitely contrived (it was asked here so no problem), but in a language like C, which I assume would evaluate such an expression on a first come basis, this behaviour can be demonstrated, if you just do `f(1) + f(0)` instead of `f(0) + f(1)`, if you keep a static variable inside of it and use it in calculating the return value of `f`.

But the question is, why would anyone do this?

Anon

The simplest solution to me is based on an observation that the whole expression is essentially supposed to evaluate to the first argument given to `f`. i.e. when evaluating from the left, we start with 0 and get 0 in the end; when evaluating from the right, we start with 1 and get 1 in the end.

Therefore, we can just return 0 if `f` has never been called (rendering this call meaningless in the addition expression), and the argument given in the previous call otherwise.

```
(define f (let ((second-last-call 0)
                 (last-call 0))
            (lambda (n)
              (set! second-last-call last-call)
              (set! last-call n)
              (+ second-last-call last-call))))
```

```
second-last-call)))
```

Tianxiang Xiong

aaa's solution is incorrect.

pluies' solution is nearly correct, except that `set!` does not return a value.

The following makes use of the simple observation that we can solve the problem if we return the argument of `f` if we have not called it before and 0 if we have previously called it.

```
(define f
  (let ((called #f))
    (lambda (x)
      (if called
          0
          (begin
            (set! called #t)
            x)))))
```

shyam

Exercise 3.8. When we defined the evaluation model in section 1.1.3, we said that the first step in evaluating an expression is to evaluate its subexpressions. But we never specified the order in which the subexpressions should be evaluated (e.g., left to right or right to left). When we introduce assignment, the order in which the arguments to a procedure are evaluated can make a difference to the result. Define a simple procedure `f` such that evaluating `(+ (f 0) (f 1))` will return 0 if the arguments to `+` are evaluated from left to right but will return 1 if the arguments are evaluated from right to left.

Solution:

```
(define (g y)
  (define (f x)
    (let ((z y))
      (set! y x)
      z)))
(define f (g 0))
```

The question is given in the context of explaining how the introduction of assignment in procedure produce differences in result when the same procedure is evaluated repeatedly with arguments in different order. i.e to show that calling `f` twice with arguments in the order `(f 0)` and then `(f 1)` is different from `(f 1)` and `(f 0)`.

Anyway it is pretty obvious that we are to design `f` with a local state variable in it ;-)

We can write the function in many ways to make the local state variable give the desired result for the above two calls. I Wrote it the way that the local state variable introduces a delayed response. procedure `g` is used to set the local environment for `f`. So we can initialize the local state variable as the argument to `g`. In this case `g` returns the desired `f` when called with the argument 0

In order to simulate the left to right and right to left evaluation, we can change the order of arguments to the given `+` as shown below. It is evaluated in MIT GNU/Scheme. It is seen to be evaluating from right to left Just to be clear, `g` and `f` are reevaluated after each experiment.

```
1 ]=>
;Value: g

1 ]=>
;Value: f

1 ]=> (+ (f 1) (f 0))

;Value: 0

1 ]=>
;Value: g

1 ]=>
;Value: f

1 ]=> (+ (f 0) (f 1))

;Value: 1
```

To be clearly out of doubt see this:

```

1 ]=>
;Value: g

1 ]=>
;Value: f

1 ]=> (f 0)
;Value: 0

1 ]=> (f 1)
;Value: 0

1 ]=>
;Value: g

1 ]=>
;Value: f

1 ]=> (f 1)
;Value: 0

1 ]=> (f 0)
;Value: 1

1 ]=>

```

pluies

```

(define f
  (let ((init (- 1)))
    (lambda (x) (if (= init (- 1))
                    (set! init x)
                    0))))

```

aaa

```

(define f
  (let ((init (/ 1 2)))
    (lambda (x)
      (set! init (- x init))
      init)))

```

MathieuBorderé

```

(define f
  (let ((state 0))
    (lambda (arg)
      (begin state (set! state arg)))))

;; testing - rename to avoid confusion
(define g
  (let ((state 0))
    (lambda (arg)
      (begin state (set! state arg)))))

(+ (g 0) (g 1))
Value: 1

(define h
  (let ((state 0))
    (lambda (arg)
      (begin state (set! state arg)))))

(+ (h 1) (h 0))
Value: 0

```

Dumb solution :-)

```

(define left_to_right 666)

(define (f x)
  (cond ((= x 0) (begin (set! left_to_right 999)
                         0))
        ((and (= x 1) (= left_to_right 666)) 0)
        (else left_to_right)))

```

```
((and (= x 1) (= left_to_right 999)) 1)))
```

GP

My solution

```
(define f
  (let ((x 1))
    (lambda (y)
      (if (= x 1)
          (begin (set! x (+ x 1)) y)
          0))))
```

musungujim

Simple back and forth state switching. Reusable in the context of a sub-expression of 2 argument addition. Odd number of calls switches the initial state.

```
(define f
  (let ((state 0))
    (lambda (x)
      (if (= state 1)
          (begin (set! state 0)
                 state)
          (begin (set! state 1)
                 x)))))
```

x3v

Exercise illustrates the potential dangers which can arise from assignment.

```
(define f
  (let ((x 0))
    (lambda (y)
      (if (> y x)
          (set! x y)
          x)))))

;; Scheme evaluates arguments from right to left
(+ (f 0) (f 1)) ;; 1
(+ (f 1) (f 0)) ;; 0
```

pat

This behavior can also simply arise from caching values, starting with 0 cached, and then returning the cached value and resetting each time.

```
(define f
  (let ((cached 0))
    (lambda (new)
      (let ((return-value cached))
        (set! cached new)
        return-value))))
```

haha

My solution

```
(define ff
  (let ((first #t))
    (lambda (s)
      (define r 0)
      (if (and first (= s 1))
          (set! r 1)
          (set! first (not first))
          r))))
```



# sicp-ex-3.9

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

[<< Previous exercise \(3.8\) | Index | Next exercise \(3.10\) >>](#)

## RECURSIVE VERSION

```
global          _____  
env           | other var.      |  
----->| factorial : *       |  
      |           |  
      |_____|_____|  
      |           |  
      |           |  
variables : n  
body: (if (= n 1) 1 (* n (factorial (- n 1))))
```

(factorial 6)

```

E1 -->| n : 6 | _____ | GLOBAL
      |
      +-----+
      (* 6 (factorial 5))
      ^
E2 -->| n : 5 | _____ | GLOBAL
      |
      +-----+
      (* 5 (factorial 4))
      ^
E3 -->| n : 4 | _____ | GLOBAL
      |
      +-----+
      (* 4 (factorial 3))
      ^
E4 -->| n : 3 | _____ | GLOBAL
      |
      +-----+
      (* 3 (factorial 2))
      ^
E5 -->| n : 2 | _____ | GLOBAL
      |
      +-----+
      (* 2 (factorial 1))
      ^
E6 -->| n : 1 | _____ | GLOBAL
      |
      +-----+

```

## ITERATIVE VERSON

```

E1 -->| n : 6 | _____ | GLOBAL
      |
      -----
      (fact-iter 1 1 n)

E2 -->| product    : 1           ^
      | counter   : 1           |
      | max-count : 6           |
      (fact-iter 1 2 6)         |

E3 -->| product    : 1           ^

```

```
| counter    : 2      ____| GLOBAL
| max-count : 6
  (fact-iter 2 3 6)

E4 -->| product    : 2      ^
| counter    : 3      ____| GLOBAL
| max-count : 6
  (fact-iter 6 4 6)

E5 -->| product    : 6      ^
| counter    : 4      ____| GLOBAL
| max-count : 6
  (fact-iter 24 5 6)

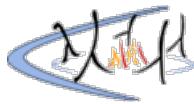
E6 -->| product    : 24     ^
| counter    : 5      ____| GLOBAL
| max-count : 6
  (fact-iter 120 6 6)

E7 -->| product    : 120    ^
| counter    : 6      ____| GLOBAL
| max-count : 6
  (fact-iter 720 7 6)

E8 -->| product    : 720    ^
| counter    : 7      ____| GLOBAL
| max-count : 6
  720
```

---

Last modified : 2018-07-18 12:18:18  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-3.10

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.9) | Index | Next exercise (3.11) >>

Does this make sense?

```
(define W1 (make-withdraw 100))
```

When `make-withdraw` is evaluated, `E0` is created with Frame A having the initial-mount binding. Next, as a result of the evaluation of the anonymous function (generated by the set (Edit: let?) structure), Frame B is created with the binding of balance (`E1` is the pointer to this frame).

```
global->| make-withdraw : * |
env.   | W1 : *      | |
-----|---^---|---^-
      | |      | |
      | |      parameter: initial-mount
      | |      body: ((lambda (balance) (...)) initial-mount)
      |
      | |____Frame_A_____
      | | initial-mount : 100 |<- E0
      | -^-----|
      |
      | |____Frame_B_____
      | | balance : initial-mount | <- E1
      | -^-----|
      |
parameter: amount
body: (if (>= balance amount) ... )
```

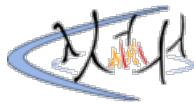
```
(W1 50)
```

`Set!` will affect Frame B, initial-mount remains unchanged in Frame A.

```
global->| make-withdraw : * |
env.   | W1 : *      | |
-----|---^---|---^-
      | |      | |
      | |      parameter: initial-mount
      | |      body: ((lambda (balance) (...)) initial-mount)
      |
      | |____Frame_A_____
      | | initial-mount : 100 |<- E0
      | -^-----|
      |
      | |____Frame_B_____
      | | balance : 50      | <- E1
      | -^-----|
      |
parameter: amount
body: (if (>= balance amount) ... )
```

Amy

It's interesting to see that the redundant lambda expression created a redundant frame. The `W1` procedure will only interact with the second created frame that has the value of `balance`.



# sicp-ex-3.11

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.10) | Index | Next exercise (3.12) >>

```
(define acc (make-account 50))

global
env -->| make-account :*
| acc : *
-----|-----^-----^-
|   |
|   ( * , * )
|   |
|   parameter: balance
|   body: (define (withdraw ... ))
|   |
|   -----Frame 0-      (parameter, body)
|   | balance : 50 |
|   E0->| withdraw : *--|---> ( * , * )
|   | deposit : *--|---> ( * , * )
|   | dispatch : *--|---> ( * , * )
|   |
|   |
|   ( * , * )
|   |
|   parameter : m
|   body      : (cond ((eq? m ... )))

((acc 'deposit) 40)
```

Frame 1 is created when (acc 'deposit is evaluated).  
Next, Frame 2 is created when (deposit amount). Since deposit is defined  
in E0, Frame 2 pointer is to environment E0.

```
global
env -->| make-account :*
| acc : *
-----|-----^-
|   |
|   -----Frame 0-
|   | balance : 50 |
|   ( *, *-)----->| withdraw : * |
|   | deposit : * |<- E0
|   | dispatch : * |
|   |
|   -----Frame 1-
|   | m : 'deposit |<- E1
|   |
|   -----Frame 2-
|   | amount : 40 |<- E2
|   |
|   ----- (deposit amount)
```

After ((acc 'deposit) 40) evaluation balance is set to 90 in Frame 0 and  
Frames 1 and 2 are not relevant anymore.

```
global
env -->| make-account :*
| acc : *
-----|-----^-
|   |
|   -----Frame 0-
|   | balance : 90 |
|   ( *, *-)----->| withdraw : * |
|   | deposit : * |<- E0
|   | dispatch : * |
|   |
|   -----
```

```
((acc 'withdraw) 60)
```

```
global
env -->| make-account :*
| acc : *
-----|-----^-
```

```

| |
| |-----Frame 0-
| | balance : 90 |
( *, **)->| withdraw : * |
| | deposit : * |<- E0
| | dispatch : * |
| |-----^----- (make-account balance)
| |
| -----Frame 3-
| | m : 'withdraw |<- E3
| |----- (dispatch m)
-----Frame 4-
| amount : 60 |<- E4
----- (withdraw amount)

```

After ((acc 'withdraw) 60)

```

global
env -->| make-account :*
| acc : *
-----| -----
| |
| |-----Frame 0-
| | balance : 30 |
( *, **)->| withdraw : * |
| | deposit : * |<- E0
| | dispatch : * |
-----|

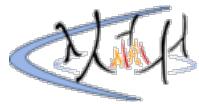
```

**aos** The above answer is great. I would just like to add an answer to the follow-up questions:

- How are the local states for the two accounts kept distinct? Which parts of the environment structure are shared between acc and acc2?
1. The local states for the two accounts each have their own environments.
  2. The only part of the environment structure that is shared between 'acc' and 'acc2' is the global environment.

**djrochford**

My only (not super important) complaint is that, in the first picture, `acc`, in the global frame, should be pointing to the same procedure object as `dispatch` is pointing to, over in Frame 0.



# sicp-ex-3.12

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

[<< Previous exercise \(3.11\)](#) | [Index](#) | [Next exercise \(3.13\) >>](#)

---

meteorgan

z  
=> (a b c d)

(cdr x)  
=> (b)

w  
=> (a b c d)

(cdr x)  
=> (b c d)

---

Last modified : 2021-08-24 18:50:12  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-3.13

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

<< Previous exercise (3.12) | Index | Next exercise (3.14) >>

---

Anon

ASCII like it's 1985

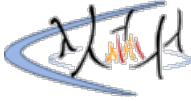
```
; ,-----,
; |
; v
; ( . ) -> ( . ) -> ( . )
; |       |       |
; v       v       v
; 'a       'b       'c
```

meteorgan

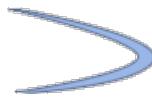
procedure make-circle will make a circular list. The **cdr** of last cell of the **list**, instead of pointing to nil points to the first cell of the list. **if** we try to compute (last-pair z) will produce infinite recursion.

---

Last modified : 2017-02-19 17:05:29  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-3.14



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

[<< Previous exercise \(3.13\) | Index | Next exercise \(3.15\) >>](#)

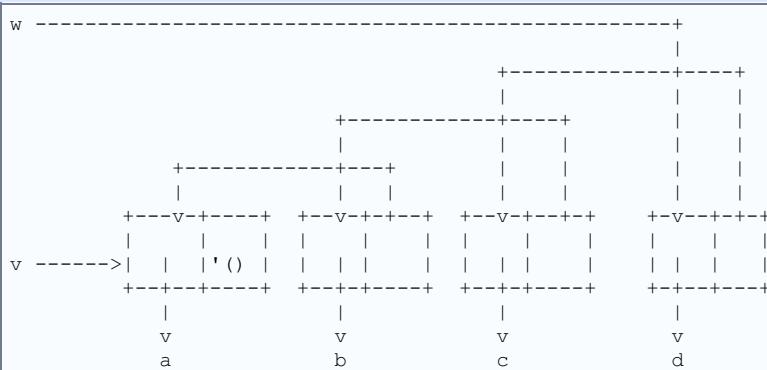
x3v

Apologies if diagram is unclear.

Below is the final box and pointer diagram after the procedure mystery is called.

W will point at the box pointing at d, whose cdr will point at the box pointing at c, so on and so forth until it reaches the box pointing at a, whose cdr points at '(). Therefore, W is now the reversed order of V.

V still points at the box pointing at a, but the cdr of a was set to be '() in the first iteration of loop. Therefore, V is now (a).



meteorgan

The procedure mystery will produce the inverse order of x.

54

$\Rightarrow$

$\Rightarrow (d \in b \setminus a)$

Bather Iffy

```
Call : mystery '(a b c d) ::
```

Result after 1 cycle through loop

```
+-----+  
| temp: . |  
+-----+  
  
-----  
| v |  
+---+---+ | +---+---+ | +---+---+ | +---+---+  
| . | / | . | .--->| . | .--->| . | / |  
+--|+---+ | +---+---+ | +--|+---+ | +--|+---+  
| v | | v | | v | | v |  
+---+ | +---+ | +---+ | +---+ | +---+ | +---+  
| a | | b | | c | | d |  
+---+ | +---+ | +---+ | +---+ | +---+ | +---+
```

Result after 2 cycle

```
+-----+  
| x: . |-----+  
| y: . |-----+  
+-----+  
  
-----  
| v | | v | | v | | v |  
+---+---+ | +---+---+ | +---+---+ | +---+---+  
| . | / | . | .--->| . | .--->| . | / |  
+--|+---+ | +---+---+ | +--|+---+ | +--|+---+  
| v | | v | | v | | v |  
+---+ | +---+ | +---+ | +---+ | +---+ | +---+  
| a | | b | | c | | d |  
+---+ | +---+ | +---+ | +---+ | +---+ | +---+
```

Result after 3 cycle

```
+-----+  
| x: . |-----+  
| y: . |-----+  
+-----+  
  
-----  
| v | | v | | v | | v |  
+---+---+ | +---+---+ | +---+---+ | +---+---+  
| . | / | . | .--->| . | .--->| . | / |  
+--|+---+ | +---+---+ | +--|+---+ | +--|+---+  
| v | | v | | v | | v |  
+---+ | +---+ | +---+ | +---+ | +---+ | +---+  
| a | | b | | c | | d |  
+---+ | +---+ | +---+ | +---+ | +---+ | +---+
```

Result after 4 cycle

```
+-----+  
| x: / |-----+  
| y: . |-----+  
+-----+  
  
-----  
| v | | v | | v | | v |  
+---+---+ | +---+---+ | +---+---+ | +---+---+  
| . | / | . | .--->| . | .--->| . | / |  
+--|+---+ | +---+---+ | +--|+---+ | +--|+---+  
| v | | v | | v | | v |  
+---+ | +---+ | +---+ | +---+ | +---+ | +---+  
| a | | b | | c | | d |  
+---+ | +---+ | +---+ | +---+ | +---+ | +---+
```

```

      v          v          |          v          |
+---+---+ +---+---+ | +---+---+ +---+---+
| . | / | . | .--- | | . | .--- | | . | .--- |
+-+---+ +---+---+ | +---+---+ | +---+---+ | +---+---+
      v          v          v          v
+---+ +---+ +---+ +---+
| a | | b | | c | | d |
+---+ +---+ +---+ +---+

```

Result after 5 cycle

Printed value :  
w : (d c b a)

Sphinxsky

meteorgan is wrong. The interpreter just made a shallow copy when it passed mystery parameters, so statement “(set-cdr! x y)” affects the value of v.

v => (a) w => (d c b a)

master

I don't really get it, why is<sub>v</sub> modified in the first iteration of<sub>loop</sub> but not the second?

intarga

Because in the second iteration, v is passed in as the second argument, not the first. Only the first argument is mutated.



# sicp-ex-3.15

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

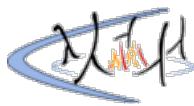
<< Previous exercise (3.14) | Index | Next exercise (3.16) >>

Anon

```
; z1 -> ( . )
;   |
;   v v
; x --> ( . ) -> ( . ) -> null
;   |
;   v         v
;   'wow      'b

; z2 -> ( . ) -> ( . ) -> ( . ) -> null
;   |
;   v         v         v
;   'a         'b
;   ^
;   |
;   -----> ( . ) -> ( . ) -> null
;   |
;   v
;   'wow
```

Last modified : 2017-02-19 22:32:46  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-3.16

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.15) | Index | Next exercise (3.17) >>

Anon

Now with more ASCII art!

```
(define (count-pairs x)
  (if (not (pair? x))
    0
    (+ (count-pairs (car x))
        (count-pairs (cdr x)))
    1)))

(define str1 '(foo bar baz))
(count-pairs str1) ; 3
; str1 -> ( . ) -> ( . ) -> ( . ) ->null
;           |           |           |
;           v           v           v
;           'foo         'bar         'baz

(define x '(foo))
(define y (cons x x))
(define str2 (list y))
(count-pairs str2) ; 4
; str2 -> ( . ) -> null
;           |
;           v
;           ( . )
;           | |
;           v v
;           ( . ) -> 'null
;           |
;           v
;           'foo

(define x '(foo))
(define y (cons x x))
(define str3 (cons y y))
(count-pairs str3) ; 7
; str3 -> ( . )
;           |
;           v v
;           ( . )
;           | |
;           v v
;           ( . ) -> null
;           |
;           v
;           'foo

(define str4 '(foo bar baz))
(set-cdr! (cddr str4) str4)
(count-pairs str4) ; maximum recursion depth exceeded
;           ,-----,
;           |
;           v
; str4 -> ( . ) -> ( . ) -> ( . )
;           |           |           |
;           v           v           v
;           'foo         'bar         'baz
```

```
(count-pairs (list 'a 'b 'c)) ; ; => 3

(define second (cons 'a 'b))
(define third (cons 'a 'b))
(define first (cons second third))
(set-car! third second)
(count-pairs first) ; ; => 4
```

```
(define third (cons 'a 'b))
(define second (cons third third))
(define first (cons second second))
(count-pairs first) ;=> 7

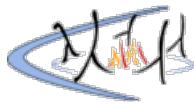
(define lst (list 'a 'b 'c))
(set-cdr! (cddr lst) lst)
(count-pairs lst) ; never returns
```

roy-tobin

Another structure for the "return 4" case.

```
(define q '(a b))
(define r4 (cons q (cdr q)))
(count-pairs r4) ; 4
; r4 -> ( . )
;           | |
;           | +----+
;           v         v
; ( . )-->( . )-> null
;           |         |
;           'a         'b
```

Last modified : 2023-02-18 00:02:30  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-3.17

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

The text suggests that one should use a structure to keep track of pairs that have already been encountered. Notice that there is a `let` statement defining `encountered` around the internal definition of a helper function, so that this list is recreated every time `count-pairs` is invoked, and so that the helper has access to all encountered pairs at any point of execution.

```
(define (count-pairs x)
  (let ((encountered '()))
    (define (helper x)
      (if (or (not (pair? x)) (memq x encountered))
          0
          (begin
            (set! encountered (cons x encountered))
            (+ (helper (car x))
                (helper (cdr x))
                1))))
    (helper x)))
```

[\*\*<< Previous exercise \(3.16\)\*\*](#) | [\*\*Index\*\*](#) | [\*\*Next exercise \(3.18\) >>\*\*](#)

aThird

A slightly different way of doing it. Note that unlike the above solution this one continues to traverse pairs it's already seen and so will never return if there's a loop.

```
(define (count-pairs x)
  (let ((counted '()))
    (define (uncounted? x)
      (if (memq x counted)
          0
          (begin
            (set! counted (cons x counted))
            1)))

    (define (count x)
      (if (not (pair? x))
          0
          (+ (count (car x))
              (count (cdr x))
              (uncounted? x))))
    (count x)))
```

antbbn

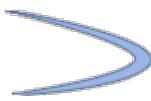
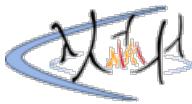
Posting my attempt as an alternative that does not use `set`

```
(define (count-pairs x)
  (define (collect-pairs x seen)
    (if (or (not (pair? x)) (memq x seen))
        seen
        (let ((seen-car (collect-pairs (car x) (cons x seen))))
          (collect-pairs (cdr x) seen-car))))
  (length (collect-pairs x '()))))
```

denis manikhin

antbbn THANK YOU VERY MUCH!!! )

# sicp-ex-3.18



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.17) | Index | Next exercise (3.19) >>

Exercise 3.18: Write a procedure that examines a list and determines whether it contains a cycle, that is, whether a program that tried to find the end of the list by taking successive cdrs would go into an infinite loop. Exercise 3.13 constructed such lists.

anonymous

```
(define (contains-cycle? lst)
  (let ((encountered (list)))
    (define (loop lst)
      (if (not (pair? lst))
          false
          (if (memq lst encountered)
              true
              (begin (set! encountered (cons lst encountered))
                     (or (loop (car lst))
                         (loop (cdr lst)))))))
    (loop lst)))
```

anonymous 2

I think the solution above is not correct eg:

```
(define t1 (cons 'a 'b))
(define t2 (cons t1 t1))
```

(cdr t2) ==> (a . b) (cdr (cdr (t2))) ==> b (contains-cycle? t2)==> #t

```
(define (contains-cycle? x)
  (define (inner return)
    (let ((C '()))
      (define (loop lat)
        (cond
          ((not (pair? lat)) (return #f))
          (else
            (if (memq (car lat) C)
                (return #t)
                (begin
                  (set! C (cons (car lat) C))
                  (if (pair? (car lat))
                      (or (contains-cycle? (car lat))
                          (loop (cdr lat)))
                      (loop (cdr lat)))))))
        (loop x))
      (call/cc inner)))
```

mbndrk

fails (goes into infinite loop) when you call (contains-cycle (car lat)) and in that sublist there's a pointer to some pair which was before that call, i. e. you are in pair A, its car points to a symbol (let it be 'a'), then you cons this symbol to the C (another potential mistake when you will eventually synchronize some entry with container C, keep in mind that (eq?) always gives #t for equal symbols since there's no way to mutate a symbol, however it's absolutely legit if any of your next boxes will point to the different 'a', but yours will break and claim it's an infinite loop which is not true), move to next pair B, where car points to the named compound sublist, and now here the (contains-cycle (car lat)) will be evaluated, but nothing accumulated in C will be passed to that call, there will be new separate frame with empty C => infinite loop.

gws

here is a simpler solution

```
(define (cycle? x)
  (define visited nil)
```

```

(define (iter x)
  (set! visited (cons x visited))
  (cond ((null? (cdr x)) false)
        ((memq (cdr x) visited) true)
        (else (iter (cdr x)))))
  (iter x))

```

Rptx

This last one is good. I would just add a clause to check if it is not a pair, to avoid an error on the cdr.

mbndrk

Yep this is a good one but not really correct. Look at this example:

```

(define x '(a b c))
(define y '(d e f))
(set-car! (cdr x) y)
(set-car! x (cdr x))
(set-cdr! (last-pair y) (cdr y))
;list y is now part of x
;here (cycle? x) gives false, but really x is an infinite loop, because it will get stuck
in (cadr x) pointer which contains the implicit loop in y, thus (caddr x) will never be
evaluated.

```

If we expect this operation to work on arbitrary list, which may contain nested lists like here, the task becomes lot more difficult. The main function of this algorithm is to find a loop in either car or cdr of the observed object, and the two conditions that satisfy this result are: 1) when either the car-pointer or the cdr-pointer point to some "box" which we've met before (all of them must be contained in a special local storage variable); 2) we can reach the current observed box from that box successively 'cdr-ing' it down; Another issue is a bit more about technical side and concerns our choice of programming style - functional or imperative way in part of organization of that "local storage" for traversed steps, albeit insignificantly change our code, wrong choice may result in a mistake and as a consequence give the wrong answer. The problem is the consequence of the way how we theoretically have to fill our "storage" and what we count as a 'infinite list' -- infinity arises only when the current node points to some predecessor node that is *on that level or the level above*, if we imagine our lists as a box-and-pointer structure with possibly more than one level, like the one pictured in the book in the figure 3.16. At the point when we reach the end of such "nested" list (the enclosing null pointer), where no cycle was found, we have to continue "walking through" the level above it, but all that stuff accumulated in the "local storage" after walking down the sublist is now a garbage and shall not be referenced anymore, because from now on any of the boxes on the main level can point to any box from that sublist (because it's on lower level and is finite, which means it doesn't contain an infinite loop). In other words, we have to throw into our "local storage" only the pointer to that sublist, not all the items it contains. However, due to the recursive structure of calls to our procedure and the fact that such sublists will also be processed with the same procedure and the same input pointer to the bound "local storage", in case of mutating that storage we will have all named garbage in there, ready for processing in further calls thus giving the wrong result. It's obscure how to deal with it in Scheme in imperative style by now. Probably introducing some additional counter variable would help, but it's so much easier just to write it the functional way. I've implemented this idea in this code and it seems to work correct for various infinite/normal lists:

```

(define (inf-loop? L)
  (define (iter items trav)
    (cond ((not (pair? items)) #f)
          ((eq? (cdr items) items) #t)
          ((eq? (car items) (cdr items))
           (iter (cdr items) trav))
          ((element-of-set? (car items) trav) #t)
          ((element-of-set? (cdr items) trav) #t)
          (else
            (if (not (pair? (car items)))
                (iter (cdr items) (cons items trav))
                (or (iter (car items) (cons items trav))
                    (iter (cdr items) (cons items trav)))))))
    (iter L '())))

```

AThird

The solutions above all seem a little too complex. After looking at them I had to recheck the question a few times to make sure I'd not missed a requirement to use mutable data structures, but I don't see any such limitation.

```

(define (has-cycle? l)
  (define (detect pair countedList)
    (cond ((not (pair? pair)) #f)
          ((memq pair countedList) #t)
          (else (detect (cdr pair) (cons pair countedList)))))
  (detect l '()))

```

fubupc

I think mbndrk' solution is the only one correct. Other solutions have mainly 2 kinds of error:

1. not consider \*car\* which actually can lead to loop as well.

2. using ONLY ONE common storage to save pairs met/counted before. e.g. if \*car\* and \*cdr\* both point to a pair P. we start by following \*car\* so P will be save into the common storage, but after that when we follow \*cdr\* P will be considered already met before so conclude that the list has circle which obviously not correct.

Following is my solution:

```
(define (has-cycle? seq)

  (define (lst-in? lst records)
    (cond ((null? records) false)
          ((eq? (car records) lst) true)
          (else (lst-in? lst (cdr records)))))

  (define (has-cycle-1? processed lst)
    (cond ((not (pair? lst)) false)
          ((lst-in? lst processed) true)
          (else
            (or (has-cycle-1? (cons lst processed) (car lst))
                (has-cycle-1? (cons lst processed) (cdr lst))))))

  (has-cycle-1? '() seq))
```

karthikk

It seems to be fubupc is exactly right. The basic idea must be that whenever we branch into the car or the cdr (tree recurse) we equip the search with a "history" of all nodes visited from the root to the nodes we are moving to so that at any point a cycle (a traversal into a node in the history) can be detected. (Now the difference between 'list' and 'list structure' could be stressed to interpret the question as strictly about loops in successive cdrs in a 'list' but the above solution will more generally detect loops in list structures and therefore to my taste preferable)

(Also note fubups's solution could be simplified by using the library function memq instead of the locally defined lst-in? : Here is a simpler version of her/his solution:

```
(define (has-loop? lis)
  (define (iter searchlist seen)
    (cond ((not (pair? searchlist)) #f)
          ((memq searchlist seen) #t)
          (else (or (iter (car searchlist) (cons searchlist seen))
                    (iter (cdr searchlist) (cons searchlist seen)))))))
  (iter lis '()))
```

Shawn

The exercise only asks us to detect cycle in lists, not list structures. The distinction is made by the author in page 100 (2nd ed). Therefore, the solution should be straight forward:

```
(define (cycle-in-list? x)
  (let ((traversed '()))
    (define (traverse x)
      (cond ((null? x) #f)
            ((memq x traversed) #t)
            (else (set! traversed (cons x traversed))
                  (traverse (cdr x)))))
    (traverse x)))
```

Sphinxsky

I think the code can be simpler.

```
(define (is-in-it? this seg)
  (if (null? seg)
      #f
      (if (eq? this (car seg))
```

```

#t
(is-in-it? this (cdr seg)))))

(define (is-cycle? x)
  (define (rec-do x seg)
    (if (pair? x)
        (or
         (is-in-it? x seg)
         (rec-do (car x) (cons x seg))
         (rec-do (cdr x) (cons x seg)))
        #f)
    (rec-do x '())))

```

Thomas (03-2020)

For me most intuitive (and rather concise) is the following: The observation is, that if there's a cycle in the list, it should repeat at some point, so the code checks whether the list repeats itself through checking whether the cdr of the current list is the same as the list itself - if not so it checks whether 1. the cdr of the curr-list repeats itself or 2. the car of the curr-list has a cycle or 3. the cdr of the curr-list has a cycle

```

(define (cycle? L)
  (define (help curr-list)
    (if (not (pair? curr-list)) false
        (if (eq? (cdr curr-list) L)
            true
            (or (help (cdr curr-list)) (cycle? (car curr-list)) (cycle? (cdr curr-list)))))))
  (help L))

```

master

I very much respect the amount of effort people here put into this problem but I too interpreted the instructions to mean "detect whether the top-level list cycles, not whether the list contains a cycle. However, I believe my solution can be called recursively on each node of the tree to determine whether there is a cycle at any level. It's quite simple: using `count-pairs` from the previous exercise, determine the number of distinct pairs in the list, and use that as the upper bound for an iterative search through the list. If we exceed this upper bound then surely we are stuck in a cycle.

```

(define (cycle? l)
  (let ((n (count-pairs l)))
    (define (iter items i)
      (cond ((null? items) #f)
            ((> i n) #t)
            (else (iter (cdr items) (+ i 1))))))
  (iter l 0)))

```

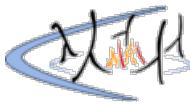
joshroybal

```

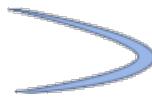
(define (is-cycle? x)
  (let ((head x))
    (define (iter y)
      (cond ((eq? y head) #t)
            ((null? y) #f)
            (else (iter (cdr y)))))
    (if (null? x)
        #f
        (iter (cdr x)))))

(define z1 (make-cycle (list 'a 'b 'c)))
(define z2 (list z1 z1))
(is-cycle? z1)
;#t
(is-cycle? z2)
;#f

```



# sicp-ex-3.19



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.18) | Index | Next exercise (3.20) >>

This is a well studied problem. Robert Floyd came up with an algorithm to solve this in the 1960s. (Yes, the same Floyd of the from the more famous Floyd-Warshall algorithm.) More information at:  
[http://en.wikipedia.org/wiki/Cycle\\_detection](http://en.wikipedia.org/wiki/Cycle_detection)

People in the software industry seem to like to use this problem as an interview question. I personally have encountered this problem in at least two separate interviews for software jobs.

The following is an implementation of Floyd's idea:

```
(define (contains-cycle? lst)
  (define (safe-cdr l)
    (if (pair? l)
        (cdr l)
        '()))
  (define (iter a b)
    (cond ((not (pair? a)) #f)
          ((not (pair? b)) #f)
          ((eq? a b) #t)
          ((eq? a (safe-cdr b)) #t)
          (else (iter (safe-cdr a) (safe-cdr (safe-cdr b)))))))
  (iter (safe-cdr lst) (safe-cdr (safe-cdr lst)))))

; Tested with mzscheme implementation of R5RS:
(define x '(1 2 3 4 5 6 7 8))
(define y '(1 2 3 4 5 6 7 8))
(set-cdr! (cdddr (cddddr y)) (cdddr y))
(define z '(1))
(set-cdr! z z)
x ; (1 2 3 4 5 6 7 8)
y ; (1 2 3 . #0=(4 5 6 7 8 . #0#))
z ; #0=(1 . #0#)
(contains-cycle? x) ; #f
(contains-cycle? y) ; #t
(contains-cycle? z) ; #t
```

This runs in linear time and constant space. Note that the space is constant because the process is iterative and the parameters and a and b are pointers -- (cdr lst) is a pointer to, not a separate copy of the rest of the list.

xdavidliu

Takes a range of size m and compares each of the nth-cdr's with the head. If no conclusions are reached, jump to the end of the range, double m, and start over.

I came up with this algorithm myself, but it appears to be similar, if not the exact same, as Brent's algorithm, which one can read about in the Cycle detection wikipedia article.

```
;
;
;
;
;

(define (cycle? x)
  (define (iter head next i m)
    (cond
      ((null? next) false) ; head is never null
      ((eq? head next) true)
      ((< i m) (iter head (cdr next) (1+ i) m))
      (else (iter next (cdr next) 1 (* 2 m)))))
    (if (null? x)
        false
        (iter x (cdr x) 1 1)))
  (cycle? '(1 2 3 4 5 6 7 8 9 10)) ; -> #f

  (define (make-cycle x)
    (set-cdr! (last-pair x) x)
    x)

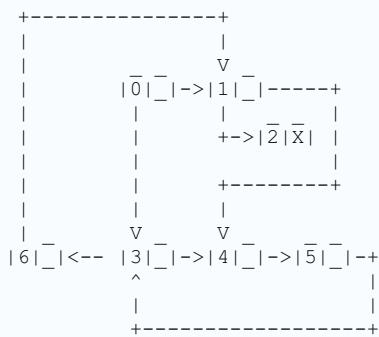
  (cycle? (make-cycle '(1 2 3 4 5 6 7 8 9 10))) ; -> #t
  (cycle? (append '(1 2 3 4 5) (make-cycle '(6 7 8 9 10 11)))) ; -> #t
```

Certainly O(1) memory, and also linear time since sum  $2^{(\lg n)} = O(n)$

HiPhish

The "Turtle and Hare" algorithm won't work for this exercise. If we take the set of nodes (pairs) to be our set S then  $f: S \rightarrow S$  has to be a function. However, note that both the car and the cdr of a pair can be pointing to another pair. This means that the "points to" relation cannot be a function from S to S, because a function can only produce one element, not two. This effectively breaks the preconditions of the algorithm and we can no longer assume that it works. In fact, the Turtle and Hare algorithm does *not* work as I am going to demonstrate.

Let's assume the following structure:



The nodes are numbered for reference, the numbers themselves are irrelevant. Node elements that point to a non-node don't have an arrow and the null element is an X. There is a cycle  $3 \rightarrow 4 \rightarrow 5 \rightarrow 3$ .

The problem starts at the first node 0 already. We can send the turtle and the hare in two directions and obviously it only makes sense to send both in the same direction. Without loss of generality we will perform a car-first search. This means we send both towards 3, at which point we move car-first towards 6, then towards 1 and from there towards 2. At this point the hare would hit a wall and conclude that that path was a dead-end.

However, in order to find the cycle the two animals would have to backtrack to the last node with a fork in the path and resume the race again from there. In other words, the algorithm needs some sort of stack, a non-constant amount of memory. I know that one could have taken another order for the search and have found the cycle, but for every order I can find a new graph where the hare will eventually run into a wall and miss the cycle.

The exercise says "Write a procedure that examines a list", so one could argue about what the term "list" in this context exactly means, but considering that the chapter has been referring to forked structures as lists as well I believe we must take forks into account as well. I don't know about an algorithm that could handle my structure or if such an algorithm even exists. Maybe this was just an oversight by the authors and I'm overthinking the issue.

Brig Says: How about just change the values in the list to 'X'. Then all you have to do is check if the current value is x or not.

Joe Replies: Your suggestion is incorrect for (list 'a 'b 'c 'X) My solution did leverage a similar idea though, and is correct given the problem description (although it does trash the original list!):

```
(define (cycle-p! list)  
  (let ((flag (cons 'doesnt 'matter)))  
    (define (f pair)  
      (cond ((null? pair) #f)  
            ((eq? flag (car pair)) #t)  
            (else (set-car! pair flag)  
                  (f (cdr pair))))  
    (f list)))
```

Solution that doesn't trash the original list:

```
(define (cycle? list)  
  (define first list)  
  (define (rec x)  
    (cond ((eq? first x) #t)  
          ((not (pair? x)) #f)  
          (else (rec (cdr x)))))  
  (rec (cdr list)))
```

David says: This solution will get stuck in an infinite loop if the first element is not in the cycle. For example

```
(define x (list 'a 'b))  
(set-cdr! (cdr x) (cdr x))
```

```
(cycle? x) ; Does not return
```

Solution that will not trash the original list(modified and recovered actually):

```
(define (cycle? x)
  (let ((ret false) (header '()))
    (define (rec lst)
      (cond ((null? lst) (set! ret false))
            ((eq? (cdr lst) header) (set! ret true))
            (else
              (let ((rest (cdr lst)))
                (set-cdr! lst header)
                (rec rest)
                (set-cdr! lst rest))))))
    (rec x)
    ret))
```

gws says: I didnt know about Floyd solution and came up with a different algorithm that is correct and constant space but it is less efficient (time-wise) and less elegant:

```
(define (cycle-const-space? x)
  (define (iter x cont elem num)
    (cond ((null? (cdr x)) false)
          ((eq? x elem) true)
          (else (if (= cont num)
                    (iter (cdr x) 0 x (+ 1 num))
                    (iter (cdr x) (+ cont 1) elem num)))))

  (iter x 0 nil 0))
```

atrika says:

I didn't know about tortoise and hare too and came up with an algorithm similar to the one of gws. It selects a node, than advance n steps. If it didn't find the selected node, it selects the current node and advance n+1 steps. It keeps increasing the number of step until it hits the empty list (no cycle) or a selected element (cycle composed of n+1 elements)

```
(define (is-cycle? mlist)
  (define (iter current tested-for remaining-steps max-steps)
    (cond ((null? current) false)
          ((eq? current tested-for) true)
          ((= remaining-steps 0) (iter (mcdr current) current (+ max-steps 1) (+ max-steps 1)))
          (else (iter (mcdr current) tested-for (- remaining-steps 1) max-steps))))
  (iter (mcdr mlist) mlist 1 1))
```

sam says: Another way is to first run (mystery x) on the list. This reversal happens in place per the definition on page256 in the book and hence constant space (that is, no extra space). If there is a cycle, 'x' (after reversal) points to the same pair as the original argument list, else no. If needed,one can recover original list by running (mystery x) again.

joe w

This seems like it works and is pretty straight forward implementation of th tortoise and hare.  
Uses constant space because cdr gets a list pointer like the explanation at the top says.  
Used racket.

```
#lang sicp
(define (last-pair x)
  (if (null? (cdr x))
      x
      (last-pair (cdr x))))

(define (make-cycle x)
  (set-cdr! (last-pair x) x)
  x)

(define z (make-cycle (list 'a 'b 'c)))
(define u (make-cycle (list 'a 'b 'c 'd 'e 'f)))
(define w (make-cycle (list 'a 'b 'c 'd 'e 'f 'g)))
(define t (make-cycle (list 'a 'b 'c 'd 'e 'f 'g 'h)))
(define v (make-cycle (list 'a 'b)))
(define x (make-cycle (list 'a)))
(define y (make-cycle (list '())))
(define a '())
(define b (list 'a))
(define c (list 'a 'b))
(define d (list 'a 'b 'c))

(define (contains-cycle? xs)
```

```

(define (chase tort hare)
  (cond ((null? hare) #f)
        ((eq? tort hare) #t)
        ((null? (cdr hare)) #f)
        (else (chase (cdr tort)
                      (cddr hare)))))

(and (pair? xs)
      (chase xs (cdr xs)))))

(contains-cycle? z)
(contains-cycle? v)
(contains-cycle? x)
(contains-cycle? y)
(contains-cycle? u)
(contains-cycle? w)
(contains-cycle? t)
(contains-cycle? a)
(contains-cycle? b)
(contains-cycle? c)
(contains-cycle? d)

```

AntonKolobov

Unfortunately, all the above solutions cannot find cycles in tree structures.

The point is to implement the DFS traversal with tracking of two periodically updatable node iterators, for finding such  $i$  and  $j$  that  $i == j$  and  $f_1(f_2(\dots f_i(\text{root})) \dots) == g_1(g_2(\dots g_j(\text{root})) \dots)$ , where  $f$  gets the next node of DFS traversal and  $g$  gets the node after the next.

```

(define (has-cycle? tree)
  ;; Helpers
  (define (iterator value idx)
    (cons value idx))
  (define (update-iterator it value idx)
    (set-car! it value)
    (set-cdr! it idx))
  (define (iterator-id it)
    (cdr it))
  (define (iterator-value it)
    (car it))
  (define (iterator-same-pos? it1 it2)
    (eq? (iterator-id it1) (iterator-id it2)))
  (define (iterator-eq? it1 it2)
    (and (iterator-same-pos? it1 it2)
         (eq? (iterator-value it1) (iterator-value it2)))))

  ;; slow-it - tracks each node (1, 2, 3, 4...)
  ;; fast-it - tracks only even nodes (2, 4...)
  (let ((slow-it (iterator tree 0))
        (fast-it (iterator '() 0))
        (clock-cnt 0))
    (define (dfs root)
      (if (not (pair? root))
          false
          (begin
            (set! clock-cnt (+ clock-cnt 1))
            (if (and (even? clock-cnt)
                     (iterator-same-pos? slow-it fast-it))
                (update-iterator slow-it root clock-cnt))
            (if (even? clock-cnt)
                (update-iterator fast-it root
                                (+ (iterator-id fast-it) 1)))
            (if (iterator-eq? slow-it fast-it)
                true
                (or (dfs (car root))
                    (dfs (cdr root)))))))
    (dfs tree)))

```

Sphinxsky

Let's start with a list:

```

(define r-inf (list 'a 'b 'c 'd))
(set-cdr! (last-pair r-inf) r-inf)

```

When the above program runs  
The position "clock-cnt" where the variable appears is as follows

```

a: 1 5 9 13 17 ... 4n+1
b: 2 6 10 14 18 ... 4n+2
c: 3 7 11 15 19 ... 4n+3
d: 4 8 12 16 20 ... 4n+4

When clock-cnt > 2 :

∴ condition: "(even? clock-cnt)"
∴ fast-it can only be "b" or "d" node
and
when fast-it is "b" node:
    fast-it-id: 1 3 5 7 9 ... 2n+1 is odd
when fast-it is "d" node:
    fast-it-id: 2 4 6 8 10 ... 2n+2 is even

The same reason
slow-it can only be "b" or "d" node too.
and slow-it-id must be even.

also ∴ condition: "(and (even? clock-cnt) (iterator-same-pos? slow-it fast-it))"
was Satisfied
    fast-it-id = slow-it-id must be even
∴ After the:
    "(update-iterator slow-it root clock-cnt)"
    "(update-iterator fast-it root (+ (iterator-id fast-it) 1))"
is executed
fast-it-id must be odd.
and slow-it-id = clock-cnt = 2 * fast-it-id = 2 * odd
∴ slow-it-id = 2 * odd
∴ slow-it must be "b" node.

∴ When slow-it-value = fast-it-value
    fast-it must be "b" node.
∴ fast-it-id: 1 3 5 7 9 ... 2n+1 is odd.
∴ slow-it-id must be even.
∴ fast-it-id != slow-it-id
∴ condition: "(iterator-eq? slow-it fast-it)" was Satisfied never.
∴ The program falls into infinite recursion when the loop section of the list is an
integer multiple of 4.

```

aQuaYi.com

Exercise 3.18: Write a procedure that examines a list and determines whether it contains a cycle, that is, whether a program that tried to find the end of the list by **taking successive cdrs** would go into an infinite loop.

notice: **taking successive cdrs**

So, we doesn't need to check loop in some car.

So, solution is very simple.

```

#lang sicp

;;; problem said, just cdr to look for end.

(define loop '(foo bar baz))
(set-cdr! (cddr loop) loop)
;
;
;
; str4 -> ( . ) -> ( . ) -> ( . )
;
;
;
;         v           v           v
;         'foo         'bar         'baz

(define (has-loop? x)
  (define (check slow fast)
    (cond ((eq? slow fast)
           #t)
          ((or (null? (cdr fast)) (null? (cddr fast)))
           #f)
          (else
            (check (cdr slow) (cddr fast)))))
    (check x (cdr x)))

(has-loop? loop)

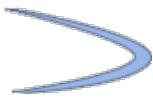
(has-loop? (list 'a 'b 'c))

```



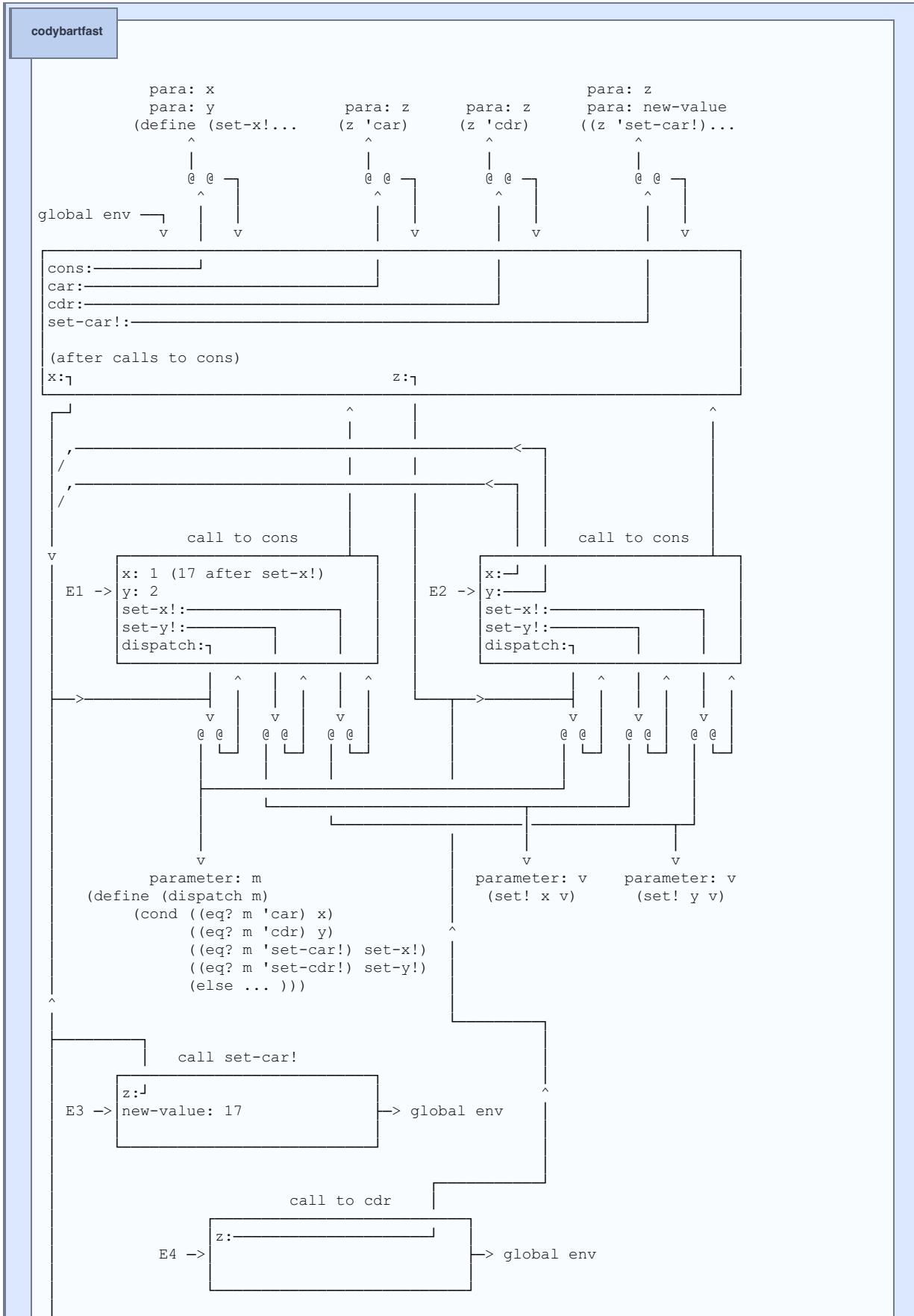


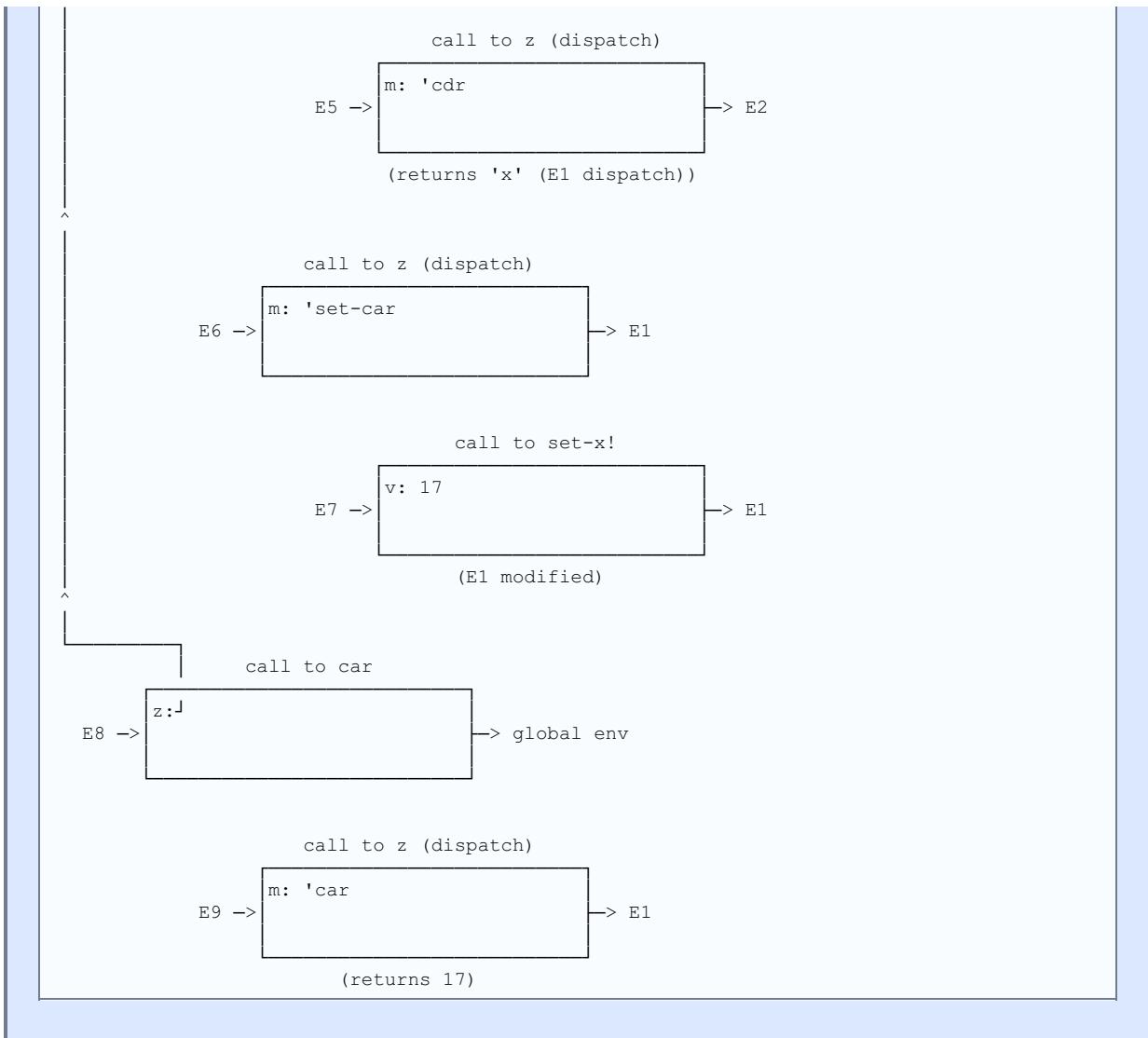
# sicp-ex-3.20



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.19) | Index | Next exercise (3.21) >>





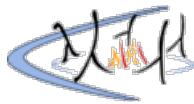
seok

What an awesome drawing by codybartfast!

I just want to point out that E5 should be E3, E3 should be E4 and E4 should be E5 in the diagram above. During the evaluation of (set-car! (cdr z) 17), the environment for set-car! is first constructed and the next is the environment for cdr. Refer to Figure 3.5 in 2nd edition for a similar case.

codybartfast

Thank you very much for detailed correction and reference to fig 3.5. I've updated the diagram above and I hope it properly incorporates your correction. Cheers!



# sicp-ex-3.21

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.21) | Index | Next exercise (3.22) >>

meteorgan

```
(define (print-queue queue) (car queue))
```

Shawn

The exercise asks us to print the queue, not return the queue. Also, we should use procedures like front-ptr instead of car. Here is my solution:

```
(define (print-queue q)
  (define (iter x)
    (if (null? x)
        (newline)
        (begin (display (car x))
                (iter (cdr x)))))
  (iter (front-ptr q)))
```

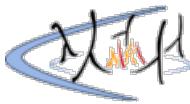
tf3

Indeed, it does ask to print, but at the end we are still printing values even if we don't explicitly invoke 'display'. You can print (+ 3 3) directly into the interpreter without a call doing (display (+ 3 3)); doing display is actually worse, it gives you ugly text like "unspecified return value" when it's done. Also, we are not 'copying' the whole queue, we are just returning a pointer, a 'cons', and we delegate the job of printing to the interpreter. So it's probably more efficient than calling display.

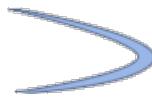
Why do you think the every queue-mutator function (in the authors' code) returns the (pointer to) queue when it's done? It's for printing the queue, or what the interpreter thinks is the representation of the queue. But us users, we don't want the explicit list structure that the queue is implemented in to be displayed. Instead of returning the (pointer to) queue we can just return (print-queue queue) in each of the functions like (insert-queue!) and (delete-queue!), that will keep bitdiddle happy

```
(define (print-queue queue) (map display (front-ptr queue)))
```

>>>



# sicp-ex-3.22



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.21) | Index | Next exercise (3.23) >>

Dewey

I suggest implement queue just like the way implement 'cons' in the section : Mutation is just assignment.

The benefit of doing in this way:

1.The expression that call of queue operation won't change. We can call them just like before:(**insert-queue!** q 'a)

2.**Keep good abstraction barrier.** Notice that the code of *empty-queue?*, *front-queue*, *insert-queue!*, *delete-queue!* don't need change at all

```
#lang sicp

;; One level of abstraction: select and to modify the front and rear pointers of a queue:
(define (make-queue)
  (let ((front-ptr '()))
    (rear-ptr '())))
(define (set-front-ptr! item)
  (set! front-ptr item))
(define (set-rear-ptr! item)
  (set! rear-ptr item))
(define (dispatch m)
  (cond ((eq? m 'front-ptr) front-ptr)
        ((eq? m 'rear-ptr) rear-ptr)
        ((eq? m 'set-front-ptr!) set-front-ptr!)
        ((eq? m 'set-rear-ptr!) set-rear-ptr!)
        (else
          (error "Undefined operation: QUEUE" m))))
  dispatch)

(define (front-ptr q) (q 'front-ptr))
(define (rear-ptr q) (q 'rear-ptr))
(define (set-front-ptr! q item)
  ((q 'set-front-ptr!) item))
(define (set-rear-ptr! q item)
  ((q 'set-rear-ptr!) item))

;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Next level abstraction: Queue operations. No change.
(define (empty-queue? queue)
  (null? (front-ptr queue)))
(define (front-queue queue)
  (if (empty-queue? queue)
      (error "FRONT called with an empty queue" queue)
      (car (front-ptr queue))))
(define (insert-queue! queue item)
  (let ((new-pair (cons item '())))
    (cond ((empty-queue? queue)
           (set-front-ptr! queue new-pair)
           (set-rear-ptr! queue new-pair)
           queue)
          (else
            (set-cdr! (rear-ptr queue) new-pair)
            (set-rear-ptr! queue new-pair)
            queue))))
(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (error "DELETE! called with an empty queue" queue))
        (else (set-front-ptr! queue (cdr (front-ptr queue)))
              queue)))

;; TEST
(define q (make-queue))
(insert-queue! q 'a) (front-ptr q);a
(front-queue q) ;a
(insert-queue! q 'b) (front-ptr q) ;a b
(delete-queue! q) (front-ptr q);b
(insert-queue! q 'c) (front-ptr q);b c
(insert-queue! q 'd) (front-ptr q);b c d
(delete-queue! q) (front-ptr q);c d
```

meteorgan

```
(define (make-queue)
  (let ((front-ptr '())
        (rear-ptr '()))
    (define (empty-queue?) (null? front-ptr))
    (define (set-front-ptr! item) (set! front-ptr item))
    (define (set-rear-ptr! item) (set! rear-ptr item))
    (define (front-queue)
      (if (empty-queue?)
          (error "FRONT called with an empty queue")
          (car front-ptr)))
    (define (insert-queue! item)
      (let ((new-pair (cons item '())))
        (cond ((empty-queue?)
               (set-front-ptr! new-pair)
               (set-rear-ptr! new-pair))
              (else
                (set-cdr! rear-ptr new-pair)
                (set-rear-ptr! new-pair))))
        (set-front-ptr! new-pair)
        (set-rear-ptr! new-pair)))
    (define (delete-queue!)
      (cond ((empty-queue?)
             (error "DELETE called with an empty queue"))
            (else (set-front-ptr! (cdr front-ptr)))))
    (define (print-queue) front-ptr)

    (define (dispatch m)
      (cond ((eq? m 'empty-queue) empty-queue?)
            ((eq? m 'front-queue) front-queue)
            ((eq? m 'insert-queue!) insert-queue!)
            ((eq? m 'delete-queue!) delete-queue!)
            ((eq? m 'print-queue) print-queue)
            (else (error "undefined operation -- QUEUE" m))))
    dispatch))
}

})})
```

genovia

```
;there is a little error in meteorgan's answer, in dispatch, should be
(empty-queue?), not empty-queue, the same as delete-queue!

(define (make-queue)
  (let ((front-ptr '()) (rear-ptr '()))
    (define (set-front-ptr! item) (set! front-ptr item))
    (define (set-rear-ptr! item) (set! rear-ptr item))

    (define (empty-queue?) (null? front-ptr))
    ;;(define (make-queue) (cons '() '()))
    (define (front-queue)
      (if (empty-queue?)
          (error "FRONT called with an empty queue" queue)
          (car front-ptr)))

    (define (insert-queue! item)
      (let ((new-pair (cons item '())))
        (cond ((empty-queue?)
               (set-front-ptr! new-pair)
               (set-rear-ptr! new-pair))
              (else
                (set-cdr! rear-ptr new-pair)
                (set-rear-ptr! new-pair)))
        front-ptr))

    (define (delete-queue!)
      (cond ((empty-queue?)
             (error "DELETE! called with an empty queue" queue))
            (else
              (set-front-ptr! (cdr front-ptr))
              front-ptr))

    (define (dispatch m)
      (cond ((eq? m 'empty-queue?) (empty-queue?))
            ((eq? m 'front-queue) (front-queue))
            ((eq? m 'insert-queue!) insert-queue!)
            ((eq? m 'delete-queue!) (delete-queue!))
            (else (error "Undefined oepration"))))

    dispatch))
}

})})
```

tf3

His answer his correct, he only has to ensure that he calls the empty-queue? method after dispatch returns it. He might have done it for the sake of consistency; your approach is also correct (and indeed more preferable).

bro\_chenzox

Here is the answer with tests

```
(define (make-queue)
  (let ((front-ptr '()))
    (rear-ptr '()))
  (define (set-front-ptr! item) (set! front-ptr item))
  (define (set-rear-ptr! item) (set! rear-ptr item))
  (define (empty-queue?) (null? front-ptr))
  (define new-queue (cons front-ptr rear-ptr))
  (define (front-queue)
    (cond ((empty-queue?) (error "FRONT-QUEUE called with an empty queue"))
          (else (car front-ptr))))
  (define (insert-queue!)
    (lambda (item)
      (let ((new-pair (cons item '())))
        (cond ((empty-queue?) (set-front-ptr! new-pair)
                           (set-rear-ptr! new-pair)
                           (cons front-ptr rear-ptr))
              (else (set-cdr! rear-ptr new-pair)
                    (set-rear-ptr! new-pair)
                    (cons front-ptr rear-ptr))))))
  (define (delete-queue!)
    (cond ((empty-queue?) (error "DELETE-QUEUE! called with an empty queue"))
          (else (set-front-ptr! (cdr front-ptr))
                (if (pair? front-ptr)
                    (cons front-ptr rear-ptr)
                    new-queue))))
  (define (dispatch m)
    (cond ((eq? m 'empty?) (empty-queue?))
          ((eq? m 'front) (front-queue))
          ((eq? m 'insert) (insert-queue!))
          ((eq? m 'delete) (delete-queue!))
          (else (error "DISPATCH called with unknown operation")))))
  (dispatch))

(define q (make-queue))
(q 'empty?) ; #t
((q 'insert) 'a) ; ((a) a)
(q 'front) ; a
(q 'empty?) ; #f
((q 'insert) 'b) ; ((a b) b)
(q 'front) ; a
(q 'delete) ; ((b) b)
(q 'delete) ; ()
```

denis manikhin

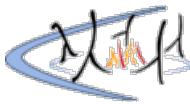
I suggest making the code shorter. Fewer letters means fewer errors.

```
#lang sicp
(define (make-queue)
  (let ((front-ptr '())
        (rear-ptr '())))
  (define (insert-q! item)
    (let ((new-pair (cons item '())))
      (cond ((null? front-ptr) (set! front-ptr new-pair)
             (set! rear-ptr new-pair) front-ptr)
            (else (set-cdr! rear-ptr new-pair)
                  (set! rear-ptr new-pair) front-ptr))))
  (define (delete-q!)
    (cond ((null? front-ptr)
           (error "DELETE! called with an empty queue" front-ptr))
          (else (set! front-ptr (cdr front-ptr)) front-ptr)))
  (define (front-q)
    (if (null? front-ptr)
        (error "FRONT called with an empty queue" front-ptr)))
```

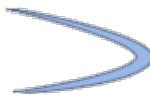
```
        (car front-ptr)))
(define (dispatch m)
  (cond ((eq? m 'insert) insert-q!)
        ((eq? m 'delete) (delete-q!))
        ((eq? m 'front) (front-q)))
        (else (error "Unknown command:" m))))
(dispatch))
```

---

Last modified : 2023-08-12 05:24:21  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-3.23



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.22) | Index | Next exercise (3.24) >>

wtative

```
;; Let 'decell' be a type of cell which contains references to two other cells
;; (e.g. next and previous cells in queue) and a value.

;; Although my 'deque' does contain a cycle, evaluating a deque itself does not force the
;; interpreter into an infinite loop, because some evaluations are delayed which is a property
;; of decell).

;----;
;;; Deque ("double-ended queue").
(define (make-deque) (cons '() '())) ; constructor

(define (empty-deque? deque)
  ; Even though an empty deque might contain
  ; a reference to an element we have "removed," we provide
  ; no access to it and consider the deque emptied.
  (or (null? (front-dptr deque))
      (null? (rear-dptr deque)))) ; predicate

(define (front-deque deque)
  (if (empty-deque? deque)
      (error "FRONT called with an empty deque" deque)
      (val-decell (front-dptr deque)))
  ; see below for definition of a decell.
(define (rear-deque deque)
  (if (empty-deque? deque)
      (error "REAR called with an empty deque" deque)
      (val-decell (rear-dptr deque)))) ; selectors

(define (set-first-deque! deque decell)
  (set-front-dptr! deque decell)
  (set-rear-dptr! deque decell))
(define (front-insert-deque! deque item)
  (let ((decell (make-decell '() item '())))
    (cond ((empty-deque? deque)
           (set-first-deque! deque decell))
          (else
            (connect-decell! decell (front-dptr deque))
            (set-front-dptr! deque decell)))
    deque))
(define (rear-insert-deque! deque item)
  (let ((decell (make-decell '() item '())))
    (cond ((empty-deque? deque)
           (set-first-deque! deque decell))
          (else
            (connect-decell! (rear-dptr deque) decell)
            (set-rear-dptr! deque decell)))
    deque))

(define (front-delete-deque! deque)
  (cond ((empty-deque? deque)
         (error "FRONT-DELETE called with an empty deque" deque))
        (else
          (set-front-dptr! deque (right-decell (front-dptr deque)))
          (if (not (empty-deque? deque))
              (set-left-decell! (front-dptr deque) '())
            deque)))
(define (rear-delete-deque! deque)
  (cond ((empty-deque? deque)
         (error "REAR-DELETE called with an empty deque" deque))
        (else
          (set-rear-dptr! deque (left-decell (rear-dptr deque)))
          (if (not (empty-deque? deque))
              (set-right-decell! (rear-dptr deque) '())
            deque))) ; mutators

(define (deque->list deque)
  (define (iter decell)
    (if (null? decell)
        '()
        (cons (val-decell decell) (iter (right-decell decell))))))
  (if (empty-deque? deque)
```

```

'()
(iter (front-dptr deque)))

;; A dequeue is a pair of front and rear references to the same list,
;; whose elements are decells.
(define (front-dptr deque) (car deque))
(define (rear-dptr deque) (cdr deque))
(define (set-front-dptr! deque decell) (set-car! deque decell))
(define (set-rear-dptr! deque decell) (set-cdr! deque decell))

;; A decell is a cell, whose car is a pair of value and
;; pointer to another decell (previous in queue). Whose cdr is a pointer
;; to another decell (next in queue).
(define (make-decell left value right)
  (cons (cons value left) right))
(define (val-decell decell) (caar decell))
(define (left-decell decell)
  (if (not (null? (cdr (car decell))))
    ;; delay/force evaluation of this part
    ;; prevents the interpreter from printing
    ;; cycle of decells.
    ((cdr (car decell)))
    '()))
(define (right-decell decell) (cdr decell))
(define (set-right-decell! decell right-decell)
  (set-cdr! decell right-decell))
(define (set-left-decell! decell left-decell)
  (set-cdr! (car decell)
            (lambda () left-decell)))

(define (connect-decell! l-decell r-decell)
  (set-left-decell! r-decell l-decell)
  (set-right-decell! l-decell r-decell))

;; Test
(define deq (make-deque))
(front-insert-deque! deq 'a)
(front-insert-deque! deq 'b)
(rear-insert-deque! deq 'z)
(rear-insert-deque! deq 'y)

(define (newline-display exp)
  (newline) (display exp))
(newline-display (dequeue->list deq))
;;Value: (b a z y)
(newline-display (front-deque deq))
;;Value: b
(front-delete-deque! deq)
(newline-display (front-deque deq))
;;Value: a
(rear-delete-deque! deq)
(newline-display (rear-deque deq))
;;Value: z

```

I've implemented the deque required in exercise 3.23 by first defining functions for doubly linked lists. I'm doing this as I'm learning Scheme so sorry for missing the idioms.

```

(define (make-dlink value prev next)
  (cons (cons value prev) next))
(define (value-dlink dlink) (car (car dlink)))
(define (next-dlink dlink) (cdr dlink))
(define (prev-dlink dlink) (cdar dlink))
(define (set-value-dlink! dlink v) (set-car! (car dlink) v))
(define (set-next-dlink! dlink ref) (set-cdr! dlink ref))
(define (set-prev-dlink! dlink ref) (set-cdr! (car dlink) ref))

(define (push-prev-dlink! dlink value)
  (and (not (null? (prev-dlink dlink)))
       (error "PUSH-PREV! called on a middle link" dlink))
  (let ((new-pair (make-dlink value null dlink)))
    (set-prev-dlink! dlink new-pair)
    new-pair))

(define (push-next-dlink! dlink value)
  (and (not (null? (next-dlink dlink)))
       (error "PUSH-NEXT! called on a middle link" dlink))
  (let ((new-pair (make-dlink value dlink null)))
    (set-next-dlink! dlink new-pair)
    new-pair))

(define (make-deque) (cons '() '()))

```

```

(define (front-ptr deque) (car deque))
(define (rear-ptr deque) (cdr deque))
(define (set-front-ptr! deque v) (set-car! deque v))
(define (set-rear-ptr! deque v) (set-cdr! deque v))

(define (empty-deque? deque) (null? (front-ptr deque)))

(define (front-deque deque)
  (if (empty-deque? deque)
      (error "FRONT called with an empty deque" deque)
      (car (front-ptr deque)))))

(define (rear-deque deque)
  (if (empty-deque? deque)
      (error "REAR called with an empty deque" deque)
      (car (rear-ptr deque)))))

(define *front* (lambda (x y) x))
(define *rear* (lambda (x y) y))

(define (insert-deque! side deque item)
  (cond ((empty-deque? deque)
         (let ((new-pair (make-dlink item null null)))
           (set-front-ptr! deque new-pair)
           (set-rear-ptr! deque new-pair)))
        (else
         (let ((push-ref-dlink! (side push-prev-dlink! push-next-dlink!))
               (ptr (side front-ptr rear-ptr))
               (set-ptr! (side set-front-ptr! set-rear-ptr!)))
           (let ((new-pair (push-ref-dlink! (ptr deque) item)))
             (set-ptr! deque new-pair))))))
  deque)

(define (front-insert-deque! deque item)
  (insert-deque! *front* deque item))

(define (rear-insert-deque! deque item)
  (insert-deque! *rear* deque item))

(define (delete-deque! side deque)
  (and (empty-deque? deque)
       (error "DELETE! called with an empty deque" deque))
  (let ((ptr (side front-ptr rear-ptr)))
    (set-ptr! (side set-front-ptr! set-rear-ptr!))
    (ref-dlink (side next-dlink prev-dlink))
    (set-ref-new-dlink! (side set-prev-dlink! set-next-dlink!))
    (set-ref-popped-dlink! (side set-next-dlink! set-prev-dlink!)))
  (let* ((pop (ptr deque))
         (new-tip (ref-dlink pop)))
    (cond ((pair? new-tip)
           (set-ref-new-dlink! new-tip null)
           (set-ptr! deque new-tip))
          (else
           (set-front-ptr! deque null)
           (set-rear-ptr! deque null)))
    (set-ref-popped-dlink! pop null)
    (value-dlink pop)))))

(define (front-delete-deque! deque)
  (delete-deque! *front* deque))

(define (rear-delete-deque! deque)
  (delete-deque! *rear* deque))

```

gws says: a more compact solution, including a print-deque function to return a list representation of the deque

```

(define (make-deque) (cons nil nil))
(define (front-ptr deque) (car deque))
(define (rear-ptr deque) (cdr deque))
(define (empty-deque? deque) (null? (front-ptr deque)))
(define (set-front! deque item) (set-car! deque item))
(define (set-rear! deque item) (set-cdr! deque item))

(define (get-item deque end)
  (if (empty-deque? deque)
      (error "Trying to retrieve item from empty deque" deque)
      (caar (end deque)))))

(define (insert-deque! deque item end)
  (let ((new-pair (cons (cons item nil) nil)))
    (cond ((empty-deque? deque)
           (set-front! deque new-pair)
           (set-rear! deque new-pair))
          ((eq? end 'front)
           (set-cdr! new-pair (front-ptr deque)))
          (else
           (set-cdr! new-pair (cdr (front-ptr deque)))))))

```

```

(set-cdr! (car (front-ptr deque)) new-pair)
(set-front! deque new-pair))
(else (set-cdr! (rear-ptr deque) new-pair)
          (set-cdr! (car new-pair) (rear-ptr deque)))
          (set-rear! deque new-pair)))))

(define (front-delete-deque deque)
  (cond ((empty-deque? deque) (error "Cannot delete from empty deque" deque))
        (else (set-front! deque (cdr (front-ptr deque))))
              (or (empty-deque? deque) (set-cdr! (car (front-ptr deque)) nil)))))

(define (rear-delete-deque deque)
  (cond ((empty-deque? deque) (error "Cannot delete from empty deque" deque))
        (else (set-rear! deque (cdar (rear-ptr deque)))
              (if (null? (rear-ptr deque)) (set-front! deque nil)
                  (set-cdr! (rear-ptr deque) nil)))))

(define (front-insert-deque! deque item) (insert-deque! deque item 'front))
(define (rear-insert-deque! deque item) (insert-deque! deque item 'rear))
(define (front-deque deque) (get-item deque front-ptr))
(define (rear-deque deque) (get-item deque rear-ptr))

(define (print-deque d)
  (define (iter res _d)
    (if (or (null? _d) (empty-deque? _d)) res
        (iter (append res (list (caaar _d))) (cons (cdar _d) (cdr d))))))
  (iter nil d))

```

danhuynh

Kinda the same as above but using internal variables instead of pair

```

;; We gonna make a two way structure like this diagram
;;           +-----+
;;           +-----+-----+
;;           +->[[a/] | -]-->[[b!] | -]>[[c!] | /]
;;
;; Which will display this in the repl of guile scheme
;; ((a) (b . #-2#) (c . #-2#))

(define (make-deque)
  (let ((front-ptr '())
        (rear-ptr '()))
    (define (dispatch m)
      (cond ((equal? m 'empty-deque?)
             (null? front-ptr))
            ((equal? m 'rear-ptr)
             rear-ptr)
            ((equal? m 'front-ptr)
             front-ptr)
            ((equal? m 'set-front-ptr!)
             (lambda (item) (set! front-ptr item)))
            ((equal? m 'set-rear-ptr!)
             (lambda (item) (set! rear-ptr item)))))

    dispatch))

(define (empty-deque? deque)
  (deque 'empty-deque?))
(define (front-ptr deque)
  (deque 'front-ptr))
(define (rear-ptr deque)
  (deque 'rear-ptr))
(define (front-deque deque)
  (if (empty-deque? deque)
      (error "FRONT called with an empty deque" deque)
      (caar (front-ptr deque))))
(define (rear-deque deque)
  (if (empty-deque? deque)
      (error "REAR called with an empty deque" deque)
      (caar (rear-ptr deque))))
(define (set-front-ptr! deque item)
  ((deque 'set-front-ptr!) item))
(define (set-rear-ptr! deque item)
  ((deque 'set-rear-ptr!) item))

(define (front-insert-deque! deque item)
  (let ((newlist (cons (cons item '()) '())))
    (if (empty-deque? deque)
        (begin
          (set-front-ptr! deque newlist)
          (set-rear-ptr! deque newlist)
          deque)
        (begin
          (set-cdr! (car (front-ptr deque)) newlist)
          (set-cdr! newlist (front-ptr deque))
          (set-front-ptr! deque newlist)

```

```

(deque) )))

(define (rear-insert-deque! deque item)
  (let ((newlist (cons (cons item '()) '())))
    (if (empty-deque? deque)
        (begin
          (set-front-ptr! deque newlist)
          (set-rear-ptr! deque newlist)
          deque)
        (begin
          (set-cdr! (car newlist) (rear-ptr deque))
          (set-cdr! (rear-ptr deque) newlist)
          (set-rear-ptr! deque newlist)
          deque)))
    (begin
      (set-cdr! (car newlist) (rear-ptr deque))
      (set-cdr! (rear-ptr deque) newlist)
      (set-rear-ptr! deque newlist)
      deque)))

(define (front-delete-deque! deque)
  (cond ((empty-deque? deque)
         (error "DELETE! called with an empty deque" deque))
        ((null? (cdar (rear-ptr deque)))
         (set-front-ptr! deque '()))
        (else
         (set-front-ptr! deque (cdr (front-ptr deque)))
         (set-cdr! (car (front-ptr deque)) '())
         deque)))

(define (rear-delete-deque! deque)
  (cond ((empty-deque? deque)
         (error "DELETE! called with an empty deque" deque))
        ((null? (cdar (rear-ptr deque)))
         (set-front-ptr! deque '()))
        (else
         (set-rear-ptr! deque (cdar (rear-ptr deque)))
         (set-cdr! (rear-ptr deque) '()))))

(define (print-deque deque)
  (display (map car
                (front-ptr deque))) (newline))

```

yooka

Use One-layer list instead of Two-layer list above and use inner define style

```

(define (make-deque)
  (let ((front-ptr '())
        (rear-ptr '()))
    ;;; sub process
    (define (front-insert-deque! item)
      (let ((new-pair (list item '() '())))
        (cond ((null? front-ptr)
               (set! front-ptr new-pair)
               (set! rear-ptr new-pair))
              (else
                (set-cdr! (cdr new-pair) front-ptr)
                (set-car! (cdr front-ptr) new-pair)
                (set! front-ptr new-pair)))
        front-ptr))
    (define (rear-insert-deque! item)
      (let ((new-pair (list item '() '())))
        (cond ((null? front-ptr)
               (set! front-ptr new-pair)
               (set! rear-ptr new-pair))
              (else
                (set-car! (cdr new-pair) rear-ptr)
                (set-cdr! (cdr rear-ptr) new-pair)
                (set! rear-ptr new-pair)))
        front-ptr))
    (define (front-delete-deque!)
      (set! front-ptr (cddr front-ptr))
      (set-car! (cdr front-ptr) '())
      front-ptr)
    (define (rear-delete-deque!)
      (set! rear-ptr (cadr rear-ptr))
      (set-cdr! (cdr rear-ptr) '())
      front-ptr)
    (define (print)
      (define (iter x result)
        (if (null? (cddr x))
            (append result (cons (car x) '()))
            (iter (cddr x) (append result (cons (car x) '())))))
      (iter front-ptr '()))

    (define (dispatch m)
      (cond ((eq? m 'front-ptr) front-ptr)
            ((eq? m 'rear-ptr) rear-ptr)
            ((eq? m 'print) print)
            ((eq? m 'empty-queue?) (null? front-ptr))
            ((eq? m 'front-delete-deque!) front-delete-deque!)
            ((eq? m 'rear-delete-deque!) rear-delete-deque!)))

```

```

        ((eq? m 'rear-insert-deque!) rear-insert-deque!)
        ((eq? m 'front-insert-deque!) front-insert-deque!)
        (else
            (display "Bad operate"))))

(dispatch)
(define mq (make-deque))
((mq 'rear-insert-deque!) 'a)
((mq 'rear-insert-deque!) 'b)
((mq 'rear-insert-deque!) 'c)
((mq 'front-insert-deque!) 'd)
((mq 'front-insert-deque!) 'e)
((mq 'front-insert-deque!) 'f)
((mq 'front-delete-deque!))
((mq 'rear-delete-deque!))
((mq 'print)))

```

GP

Since we have learnt in Chap 2. about data abstraction and abstraction layers, it makes more sense here to build one more abstraction layer by adding properly designed constructors and selectors. The solution will be more modularized and extendable. The code is also a bit more readable in my opinion.

```

; using local variables and message passing instead of defining function under global
environment

(define (make-deque)
  (let ((front-ptr '()))
    (rear-ptr '()))

  ; one more abstraction barrier allows flexible and extendable implementation for both
  lower and higher level methods
  ; one example here is modifying lower level implementation of node structure dose not
  the methods for deque
  ; also, higher level modification (modifying deque methods) will not influence lower
  level implementation

  ; lower level constructor and selectors
  ; each has two different implementations. (higher level methods do not need to be
  changed)
  (define (make-node item)
    (cons (cons item '()) '()))
  (define (make-node1 item)
    (cons item (cons '() '())))

  (define (node-item node)
    (caar node))
  (define (node-item1 node)
    (car node))

  (define (next-node node)
    (cdr node))
  (define (next-node1 node)
    (cddr node))

  (define (prev-node node)
    (cdar node))
  (define (prev-node1 node)
    (cadar node))
  (define (set-prev-node! node prevnode)
    (set-cdr! (car node) prevnode))
  (define (set-prev-node!1 node prevnode)
    (set-car! (cdr node) prevnode))
  (define (set-next-node! node nextnode)
    (set-cdr! node nextnode))
  (define (set-next-node!1 node nextnode)
    (set-cdr! (cdr node) nextnode))

  (define (set-front-ptr! item)
    (set! front-ptr item))
  (define (set-rear-ptr! item)
    (set! rear-ptr item))

  ; higher level methods for deque
  ; constructors and selectors
  (define (empty-deque?) (null? front-ptr))

  (define (front-deque)
    (if (empty-deque?)
        (error "FRONT called with an empty deque")
        (node-item front-ptr)))

```

```

(define (front-insert-deque! item)
  (let ((new-node (make-node item)))
    (cond ((empty-deque?)
           (set-front-ptr! new-node)
           (set-rear-ptr! new-node))
          (else
            (set-prev-node! front-ptr new-node)
            (set-next-node! new-node front-ptr)
            (set-front-ptr! new-node)))))

; methods to modify deques
(define (rear-insert-deque! item)
  (let ((new-node (make-node item)))
    (cond ((empty-deque?)
           (set-front-ptr! new-node)
           (set-rear-ptr! new-node))
          (else
            (set-prev-node! new-node rear-ptr)
            (set-next-node! rear-ptr new-node)
            (set-rear-ptr! new-node)))))

(define (front-delete-deque!)
  (cond ((empty-deque?)
         (error "DELETE! called with an empty deque"))
        (else
          (set-front-ptr! (next-node front-ptr))
          (if (null? front-ptr)
              (set-rear-ptr! '())
              (set-prev-node! front-ptr '())))))

(define (rear-delete-deque!)
  (cond ((empty-deque?)
         (error "DELETE! called with an empty deque"))
        (else
          (set-rear-ptr! (prev-node rear-ptr))
          (if (null? rear-ptr)
              (set-front-ptr! '())
              (set-next-node! rear-ptr '())))))

(define (print-deque)
  (define (print-node node)
    (cond ((null? node) nil)
          (else (cons (node-item node) (print-node (next-node node)))))))
  (display (print-node front-ptr))
  (newline))

(define (dispatch m)
  (cond ((eq? m 'front-deque) front-deque)
        ((eq? m 'empty-deque?) empty-deque?)
        ((eq? m 'front-insert-deque!) front-insert-deque!)
        ((eq? m 'rear-insert-deque!) rear-insert-deque!)
        ((eq? m 'front-delete-deque!) front-delete-deque!)
        ((eq? m 'rear-delete-deque!) rear-delete-deque!)
        ((eq? m 'print-deque) print-deque)))
  dispatch)

;define global methods with message passing
(define (front-deque deque) ((deque 'front-deque)))
(define (empty-deque? deque) ((deque 'empty-deque?)))
(define (front-insert-deque! deque item) ((deque 'front-insert-deque!) item))
(define (rear-insert-deque! deque item) ((deque 'rear-insert-deque!) item))
(define (front-delete-deque! deque) ((deque 'front-delete-deque!)))
(define (rear-delete-deque! deque) ((deque 'rear-delete-deque!)))
(define (print-deque deque) ((deque 'print-deque)))

;; ---- TESTING CODE ----
;test deque constructor and selectors
(define d (make-deque))
(empty-deque? d)
(print-deque d)

; testing front insert
(newline)
(display "---- testing front insert ----")
(newline)
(front-insert-deque! d 'a)
(print-deque d)
(front-insert-deque! d 'b)
(print-deque d)
(front-insert-deque! d 'c)
(print-deque d)

; testing front delete
(newline)
(display "---- testing front delete ----")
(newline)
(front-delete-deque! d)

```

```

(print-deque d)
(front-delete-deque! d)
(print-deque d)
(front-delete-deque! d)
(print-deque d)

; testing rear insert
(newline)
(display "---- testing rear insert ----")
(newline)
(rear-insert-deque! d 'a)
(print-deque d)
(rear-insert-deque! d 'b)
(print-deque d)
(rear-insert-deque! d 'c)
(print-deque d)

; testing rear delete
(newline)
(display "---- testing rear delete ----")
(newline)
(rear-delete-deque! d)
(print-deque d)
(rear-delete-deque! d)
(print-deque d)
(rear-delete-deque! d)
(print-deque d)

; testing front-insert and rear delete
(newline)
(display "---- testing front-insert and rear delete ----")
(newline)
(front-insert-deque! d 'a)
(print-deque d)
(rear-delete-deque! d)
(print-deque d)

```

tandoide My solution uses a list called rear-list to store the pairs in reversed order (note that it is not a reversed queue because the pairs are not linked, just belong to this list as elements). When an item is add either front or rear the rear-list also grows (front for rear-add and rear for front-add). When we have to remove from rear we take the car of rear-list and know what pair is "before" our rear-ptr, this pair's cdr is our rear-ptr before the delete operation. It works on O(1) time, but the space is not efficient because every time we add an element to the deque each element of the rear-list (the pairs of the deque) also grow by one element. EDIT: my solution has a bug, it's that when we remove front the front in the deque we should also remove the last element of the rear-list, but I can not think of a way in which this could be done in O(1) time.

```

(define (make-deque)
  (let ((front-ptr '()))
    (rear-ptr '())
    (rear-list '())))
(define (empty-deque?)
  (or (null? front-ptr) (null? rear-ptr)))
(define (front-insert-deque! item)
  (let ((new-pair (cons item '())))
    (cond ((empty-deque?)
            (set! front-ptr new-pair)
            (set! rear-ptr new-pair))
          (else
            (set-cdr! new-pair front-ptr)
            (set! front-ptr new-pair)
            (cond ((null? rear-list)
                    (set! rear-list
                          (append (list new-pair)
                                  rear-list)))
                  (else
                    (set! rear-list
                          (append rear-list (list new-pair))))))))
(define (rear-insert-deque! item)
  (let ((new-pair (cons item '())))
    (cond ((empty-deque?)
            (set! front-ptr new-pair)
            (set! rear-ptr new-pair))
          (else
            (set-cdr! rear-ptr new-pair)
            (set! rear-list (append (list rear-ptr) rear-list))
            (set! rear-ptr new-pair))))
(define (front-delete-deque!)
  (cond ((null? front-ptr) (set! front-ptr '()))
        ((null? (cdr front-ptr))
         (set! front-ptr '())
         (set! rear-ptr '()))
        (else
          (set! front-ptr (cdr front-ptr))))))

```

```

(define (rear-delete-deque!)
  (cond ( (null? rear-ptr) (set! rear-ptr '()))
        ( (null? rear-list)
          (set! rear-ptr '())
          (set! front-ptr '()))
        (else
          (set-cdr! (car rear-list) '())
          (set! rear-ptr (car rear-list))
          (set! rear-list (cdr rear-list))
          )))
(define (print-deque)
  (display front-ptr))
  (newline)
(define (dispatch m)
  (cond ((eq? m 'front-ptr) front-ptr)
        ((eq? m 'rear-ptr) rear-ptr)
        ((eq? m 'front-insert-deque!) front-insert-deque!)
        ((eq? m 'rear-insert-deque!) rear-insert-deque!)
        ((eq? m 'front-delete-deque!) (front-delete-deque!))
        ((eq? m 'rear-delete-deque!) (rear-delete-deque!))
        ((eq? m 'empty-deque?) (empty-deque?))
        ((eq? m 'print-deque) (print-deque))))
  dispatch))

```

master

Omitting all constructors and selectors previously defined, they were left unchanged. Maybe it's because I didn't think about the problem hard enough before getting started, but I implemented all of the deque operations using a standard list representation until I got to `rear-delete-deque!` which is of course impossible to implement that way (at least if it needs to take constant time). So I had to rewrite everything using a new representation and for that reason it may be a little bit more ad-hoc than the other solutions here but OTOH it is exactly as much mechanism as is needed, no more no less. The representation I went for is something akin to a doubly linked list, where the `cars` are all nodes which contain the symbol at that node plus a linked list which goes in the reverse direction (which I call `cirs` (contents of increment register) to keep with the terminology). It is an extension of the original list, i.e. the `cirs` are pointers to previous nodes, so there is no need to keep a reversed copy of the original list around.

```

(define (make-node cir item)
  (cons cir item))

(define (node-cir node)
  (car node))

(define (node-item node)
  (cdr node))

(define (front-insert-deque! deque item)
  (cond ((empty-deque? deque)
         (let ((new-node (make-node '() item)))
           (let ((new-pair (cons new-node '())))
             (set-front-ptr! deque new-pair)
             (set-rear-ptr! deque new-pair)
             (print-deque deque)))
         )
        (else
         (let ((new-node (make-node '() item)))
           (let ((new-pair (cons new-node '())))
             (set-car! (front-deque deque) new-node)
             (set-cdr! new-pair (front-ptr deque))
             (set-front-ptr! deque new-pair)
             (print-deque deque)))))

(define (rear-insert-deque! deque item)
  (cond ((empty-deque? deque)
         (let ((new-node (make-node '() item)))
           (let ((new-pair (cons new-node '())))
             (set-front-ptr! deque new-pair)
             (set-rear-ptr! deque new-pair)
             (print-deque deque)))
         )
        (else
         (let ((new-node (make-node '() item)))
           (let ((new-pair (cons new-node '())))
             (set-car! new-node (rear-deque deque))
             (set-cdr! (rear-ptr deque) new-pair)
             (set-rear-ptr! deque new-pair)
             (print-deque deque))))))

(define (front-delete-deque! deque)
  (cond ((empty-deque? deque)
         (error "DELETE! called with an empty deque" deque))
        (else (set-front-ptr! deque (cdr (front-ptr deque)))
              (print-deque deque))))

```

```

(define (rear-delete-deque! deque)
  (cond ((empty-deque? deque)
         (error "DELETE! called with an empty deque" deque))
        (else (set-rear-ptr! deque (node-cir (rear-ptr deque)))
              (set-cdr! (rear-ptr deque) '())
              (print-deque deque)))))

(define (print-deque deque)
  (let ((items (front-ptr deque)))
    (define (helper rest)
      (if (null? rest)
          (newline)
          (begin (if (not (null? (node-item (car rest)))))
                    (display (node-item (car rest))))
                 (display " ")
                 (helper (cdr rest))))))
    (if (cycle? deque)
        (error "Deque contains a cycle, unable to print" deque)
        (begin (helper items)
               (newline))))))

```

x3v

Tried to make as few changes to the message-passing style queue structure as possible.  
Key change is every new item contains 3 elements now: (value, leftptr, rightptr), as opposed to (value, rightptr).

```

(define (make-deque)
  (let ((front-ptr '()))
    (rear-ptr '())))
  (define (set-front-ptr! item) (set! front-ptr item))
  (define (set-rear-ptr! item) (set! rear-ptr item))
  (define (empty-deque?) (null? front-ptr))
  (define (front-deque)
    (if (empty-deque?) (error "empty queue") (car front-ptr)))
  (define (rear-deque)
    (if (empty-deque?) (error "empty deque") (car rear-ptr)))
  (define (front-insert-deque! item)
    (let ((new-pair (list item '() '())))
      (cond ((empty-deque?)
              (set-front-ptr! new-pair)
              (set-rear-ptr! new-pair))
             (else (set-car! (cddr new-pair) front-ptr)
                   (set-car! (cdr front-ptr) new-pair)
                   (set-front-ptr! new-pair))))
      front-ptr))
  (define (rear-insert-deque! item)
    (let ((new-pair (list item '() '())))
      (cond ((empty-deque?)
              (set-front-ptr! new-pair)
              (set-rear-ptr! new-pair))
             (else (set-car! (cdr new-pair) rear-ptr)
                   (set-car! (cddr rear-ptr) new-pair)
                   (set-rear-ptr! new-pair))))
      front-ptr))
  (define (front-delete-deque!)
    (cond ((empty-deque?) (error "empty deque"))
          ((eq? front-ptr rear-ptr)
           (set! front-ptr '())
           (set! rear-ptr '()))
          (else (set-front-ptr! (caddr front-ptr))
                (set-car! (cdr front-ptr) '()))))
      front-ptr))
  (define (rear-delete-deque!)
    (cond ((empty-deque?) (error "empty deque"))
          ((eq? front-ptr rear-ptr)
           (set! rear-ptr '())
           (set! front-ptr '()))
          (else (set-rear-ptr! (cadar rear-ptr))
                (set-car! (cddr rear-ptr) '()))))
      front-ptr))
  (define (print)
    (define (iter p)
      (if (null? p)
          '()
          (cons (car p) (iter (caddr p))))))
    (iter front-ptr)))
  (define (dispatch m)
    (cond ((eq? m 'front-deque) (front-deque))
          ((eq? m 'rear-deque) (rear-deque))
          ((eq? m 'empty-deque?) (empty-deque?))
          ((eq? m 'front-insert-deque!) front-insert-deque!)
          ((eq? m 'rear-insert-deque!) rear-insert-deque!)
          ((eq? m 'front-delete-deque!) (front-delete-deque!))))
```

```

((eq? m 'rear-delete-deque!) (rear-delete-deque!))
((eq? m 'print) (print))
(else (error "incorrect usage" m)))
dispatch))

(define (front-deque dq)
  (dq 'front-deque))
(define (rear-deque dq)
  (dq 'rear-deque))
(define (empty-deque? dq)
  (dq 'empty-deque?))
(define (front-insert-deque! dq item)
  ((dq 'front-insert-deque!) item))
(define (rear-insert-deque! dq item)
  ((dq 'rear-insert-deque!) item))
(define (rear-delete-deque! dq)
  (dq 'rear-delete-deque!)
  (print-dq dq))
(define (front-delete-deque! dq)
  (dq 'front-delete-deque!)
  (print-dq dq))
(define (print-dq dq)
  (dq 'print))

```

roy-tobin

Nicely done. Compact with a lovely pattern of indentation. And no pollution of the global environment. This implementation passes a 1.3k transaction **torture test** I created. Others on this page (Feb 2023) do not. My own deque solution is at the link, nothing new, but fully tested.

tch

Just use a double direction linked list like everyone else.

```

; deque: double ended queue, (front-node. rear-node)
(define (make-deque) (cons nil nil))
(define (empty-deque? q) (null? (front-node-deque q)))
(define (front-deque q) (value-node (front-node-deque q)))
(define (rear-deque q) (value-node (rear-node-deque q)))

; auxiliary procedures
(define (deque-only-one-element? q) (eq? (front-node-deque q) (rear-node-deque q)))
(define (front-node-deque q) (car q))
(define (rear-node-deque q) (cdr q))
(define (set-deque-front! q value) (set-car! q value))
(define (set-deque-rear! q value) (set-cdr! q value))

; node of deque (just like a double-direction linked list)
(define (make-node prev value next) (list 'node prev value next))
(define (prev-node node) (cadr node))
(define (value-node node) (caddr node))
(define (next-node node) (cadddr node))
(define (set-node-prev! node value) (set-car! (cdr node) value))
(define (set-node-next! node value) (set-car! (cdddr node) value))

; use a list of node to construct deque
; all return values are unspecified
(define (front-insert-deque! q value)
  (let ((new-node (make-node nil value nil)))
    (cond ((empty-deque? q) (set-deque-front! q new-node)
           (set-deque-rear! q new-node))
          (else (set-node-next! new-node (front-node-deque q))
                (set-node-prev! (front-node-deque q) new-node)
                (set-deque-front! q new-node)))
    )
  )
)
(define (rear-insert-deque! q value)
  (let ((new-node (make-node nil value nil)))
    (cond ((empty-deque? q) (set-deque-front! q new-node)
           (set-deque-rear! q new-node))
          (else (set-node-next! (rear-node-deque q) new-node)
                (set-node-prev! new-node (rear-node-deque q))
                (set-deque-rear! q new-node)))
    )
  )
)
(define (front-delete-deque! q)
  (cond ((empty-deque? q) (error "Delete front from an empty deque."))
        ((deque-only-one-element? q) (set-deque-front! q nil)
         (set-deque-rear! q nil)) ; only one element
        (else (set-deque-front! q (next-node (front-node-deque q)))
              (set-node-prev! (front-node-deque q) nil)))

```

```

        )
(define (rear-delete-deque! q)
  (cond ((empty-deque? q) (error "Delete rear from an empty deque."))
        ((deque-only-one-element? q) (set-deque-front! q nil)
         (set-deque-rear! q nil)) ; only one element
        (else (set-deque-rear! q (prev-node (rear-node-deque q)))
              (set-node-next! (rear-node-deque q) nil)))
      )
)
(define (print-deque q)
  (define (iter node)
    (if (not (null? node))
        (begin (display (value-node node))
               (display " "))
               (iter (next-node node)))
      )
    )
  (display "#deque front ... rear : ")
  (iter (front-node-deque q))
  (newline)
)

; test
(define q (make-deque))
(print-deque q)
(front-insert-deque! q 1)
(front-insert-deque! q 2)
(front-insert-deque! q 3)
(rear-insert-deque! q 10)
(rear-insert-deque! q 100)
(rear-insert-deque! q 1000)
(print-deque q)

(front-deque q)
(rear-deque q)

(front-delete-deque! q)
(rear-delete-deque! q)
(rear-delete-deque! q)
(print-deque q)

```

bro-chenzox

```

(define (make-deque)
  (let ((front-node '()) ; () ; () a () ; () a () b () ; 
        (rear-node '()) ; () ; () a () ; () a () c () ; ((( a () c ()) c () 
  (define (empty-deque?) (null? front-node))
  (define (front-deque)
    (if (empty-deque?)
        (error "FRONT-DEQUE was called in an empty deque")
        (node-value front-node)))
  (define (rear-deque)
    (if (empty-deque?)
        (error "REAR-DEQUE was called in an empty deque")
        (node-value rear-node)))

; auxiliary procedures
(define (set-front-node! item) (set! front-node item))
(define (set-rear-node! item) (set! rear-node item))
(define (single-item-deque?) (eq? front-node rear-node))

; node structure
(define (make-node prev value next) (list prev value next))
(define (node-prev node) (car node))
(define (node-value node)
  (if (null? node)
    '()
    (cadr node)))
(define (node-next node) (caddr node))
(define (set-node-prev! node item) (set-car! node item))
(define (set-node-next! node item) (set-car! (cddr node) item))

(define (insert-deque! side)
  (lambda (item)
    (let ((new-node (make-node '() item '())))
      ; () a () ; () b () ; () c () ;
      ((( a () c ()) c ())
       (cond ((empty-deque?)
              (set-front-node! new-node)
              (set-rear-node! new-node))
          (else
            (cond ((eq? side 'front)

```

```

(set-node-next! new-node front-node)
(set-front-node! new-node))
((eq? side 'rear)
  (set-node-next! rear-node new-node)
  (set-node-prev! new-node rear-node)
  (set-rear-node! new-node))))))
(cons (list (node-value front-node)) (list (node-value rear-node)))))

(define (delete-deque! side)
  (cond ((empty-deque?)
         (error "DELETE-DEQUE! called with an empty deque"))
        ((single-item-deque?)
         (set-front-node! '())
         (set-rear-node! '())
         'empty-dq)
        (else
         (cond ((eq? side 'front)
                (set-front-node! (node-next front-node))
                (set-node-prev! front-node '()))
                ((eq? side 'rear)
                 (set-rear-node! (node-prev rear-node))
                 (set-node-next! rear-node '())))
                (cons (list (node-value front-node)) (list (node-value rear-node)))))

(define (front-insert-deque!) (insert-deque! 'front))
(define (rear-insert-deque!) (insert-deque! 'rear))
(define (front-delete-deque!) (delete-deque! 'front))
(define (rear-delete-deque!) (delete-deque! 'rear))

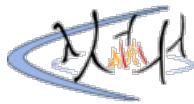
(define (dispatch m)
  (cond ((eq? m 'front) (front-deque))
        ((eq? m 'rear) (rear-deque))
        ((eq? m 'front-insert) (front-insert-deque!))
        ((eq? m 'rear-insert) (rear-insert-deque!))
        ((eq? m 'front-delete) (front-delete-deque!))
        ((eq? m 'rear-delete) (rear-delete-deque!)))
  dispatch))

(define dq (make-deque))
((dq 'front-insert) 'a)
((dq 'front-insert) 'b)
((dq 'rear-insert) 'c)

;(dq 'front-delete)
;(dq 'front)
;(dq 'rear)
;(dq 'rear-delete)
;(dq 'rear-delete)
;((dq 'rear-insert) 'a)

```

[<< Previous exercise \(3.22\)](#) | [Index](#) | [Next exercise \(3.24\) >>](#)



# sicp-ex-3.24

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.23) | Index | Next exercise (3.25) >>

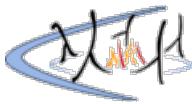
meteorgan

```
; all we need to do is changing the procedure assoc.  
(define (make-table same-key?)  
  (let ((local-table (list '*table*)))  
    (define (assoc key records)  
      (cond ((null? records) #f)  
            ((same-key? key (caar records)) (car records))  
            (else (assoc key (cdr records)))))  
    (define (lookup key-1 key-2)  
      (let ((subtable (assoc key-1 (cdr local-table))))  
        (if subtable  
            (let ((record (assoc key-2 (cdr subtable))))  
              (if record  
                  (cdr record)  
                  #f))  
            #f)))  
    (define (insert! key-1 key-2 value)  
      (let ((subtable (assoc key-1 (cdr local-table))))  
        (if subtable  
            (let ((record (assoc key-2 (cdr subtable))))  
              (if record  
                  (set-cdr! record value)  
                  (set-cdr! subtable  
                            (cons (cons key-2 value)  
                                  (cdr subtable))))  
              (set-cdr! local-table  
                        (cons (list key-1  
                                    (cons key-2 value))  
                              (cdr local-table))))))  
    (define (dispatch m)  
      (cond ((eq? m 'lookup-proc) lookup)  
            ((eq? m 'insert-proc!) insert!)  
            (else (error "Unknown operation -- TABLE" m))))  
  dispatch))  
  
(define operation-table (make-table))  
(define get (operation-table 'lookup-proc))  
(define put (operation-table 'insert-proc!))
```

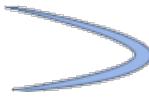
tch

everything we need is just a definition of our own assoc, a simple test

```
; put and get  
(define operation-table (make-table (lambda (x y) (< (abs (- x y)) 0.1))))  
(define get (operation-table 'lookup-proc))  
(define put (operation-table 'insert-proc))  
  
; test  
(put 1.0 1.0 'hello)  
(get 1.01 1.01)
```



# sicp-ex-3.25



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.24) | Index | Next exercise (3.26) >>

roy-tobin

Only two of the twelve solutions below pass a **torture test** as of Feb 2023. Please see the **full report** for methodology details and the findings of this analysis.

dzy

just need a little change to make the process recursive.

```
(define (make-table)
(define local-table (cons '*table* '()))
(define (get-key table) (caaar table))
(define (get-value item)
  (if (pair? (car item))
      (if (null? (cdar item))
          #f
          (cdar item))
      #f))
(define (change-value item value) (set-cdr! (car item) value))
(define (add-subtable table key value) (set-cdr! table
  (cons (cons (cons key value)
    '())
    (cdr table))))
(define (assoc key table)
  (cond ((null? table) #f)
        ((= key (get-key table)) (car table))
        (else (assoc key (cdr table)))))
(define (lookup keys)
  (define (iter keys table)
    (if (null? keys)
        (begin (display table) (get-value table))
        (let ((result (assoc (car keys) (cdr table))))
          (if result
              (iter (cdr keys) result)
              #f))))
  (iter keys local-table))
(define (insert! value keys)
  (define (iter value keys table)
    (let ((result (assoc (car keys) (cdr table))))
      (if (null? (cdr keys))
          (if result
              (change-value result value)
              (add-subtable table (car keys) value))
          (if result
              (iter value (cdr keys) result)
              (begin (add-subtable table (car keys) '())
                (iter value (cdr keys) (cadar table)))))))
  (iter value keys local-table))
(define (dispatch m)
  (cond ((eq? m 'insert!) insert!)
        ((eq? m 'lookup) lookup)
        ((eq? m 'print) (display local-table))))
(dispatch)
(define (insert! x value . keys) ((x 'insert!) value keys))
(define (lookup x . keys) ((x 'lookup) keys))
(define (print t) (t 'print))
(define (delete! t . keys) ((x 'insert!) '() keys))
(define t (make-table))
(insert! t 'a 1 1)
(lookup t 1 1)
(insert! t 'b 2)
(insert! t 'c 3)
(insert! t 'c 2 3 4)
(insert! t 'd 2 3 5)
(insert! t 'e 2 3 6)
(lookup t 3 4)
(lookup t 2 3)
(lookup t 2 3 4)
(lookup t 1)
(insert! t 'x 1 1)
(insert! t 'y 2 3 4)
(lookup t 1 1)
```

```
(lookup t 2 3 4)
(lookup t 2 3 4)
```

eric4brs

The answer posted by Genovia below fails in two respects. First you cannot insert a new value in an existing location.

```
(table-insert! mytable 'a 1 1)
(table-insert! mytable 'a 1 1)
```

The second table-insert! fails. Second you cannot add a value to a location that contains a sub-tree. That is, Genovia's node does not support both a value and a sub-tree.

```
(table-insert! mytable 'a 1 1)
(table-insert! mytable 'b 1)
```

The second table insert fails. The following solution supports both of these cases. It also supports delete. It also supports a draw function if you are lucky enough to use an implementation that supports diagramming. I've included my test cases which exercise node containing value and sub-tree, changing a value, deleting a value, and drawing table (commented in case your implementation doesn't support draw)

```
(define (make-table)
  (let ((local-table (list '*table* nil)))

    (define (assoc key records)
      (cond ((null? records) false)
            ((equal? records '()) false)
            ((equal? key (car (car records))) (car records))
            (else (assoc key (cdr records)))))

    (define (find subtable keys)
      (let ((record (assoc (car keys) (cdr (cdr subtable)))))
        (if record
            (if (null? (cdr keys))
                (list nil record)
                (find record (cdr keys)))
            (list keys subtable)))

    (define (new-branch! table keys value)
      (define (recurse keys value)
        (if (null? (cdr keys))
            (cons (car keys) (list value))
            (cons (car keys) (list nil (recurse (cdr keys) value)))))

      (if (not (pair? keys))
          #f
          (set-cdr! (cdr table) (cons (recurse keys value) (cdr (cdr table))))))

    (define (display)
      (draw local-table))

    (define (insert! keys value)
      (let ((find-result (find local-table keys)))
        (let ((subkeys (car find-result))
              (subtable (car (cdr find-result))))
          (if (null? subkeys)
              (set-car! (cdr subtable) value)
              (new-branch! subtable subkeys value)))))

      'ok)

    (define (lookup keys)
      (let ((find-result (find local-table keys)))
        (let ((subkeys (car find-result))
              (subtable (car (cdr find-result))))
          (if (null? subkeys)
              (let ((value (car (cdr subtable))))
                (if (equal? value nil)
                    #f
                    value))
              #f)))

    (define (dispatch m)
      (cond ((eq? m 'lookup) lookup)
            ((eq? m 'insert!) insert!)
            ((eq? m 'draw) (display))
            (else (error "Unknown operation -- MAKE-TABLE" m)))))

    dispatch))

  (define t-t-t-t (make-table))
  (define (table-insert! value . keys)
    ((t-t-t-t 'insert!) keys value))
  (define (table-delete! . keys)
    ((t-t-t-t 'insert!) keys nil))
  (define (table-lookup . keys)
    ((t-t-t-t 'lookup) keys))
  (define (table-draw)
```

```
(t-t-t-t 'draw))

(table-insert! 'a 1 1)
(table-lookup 1 1)
(table-insert! 'b 2)
(table-insert! 'c 3)
(table-insert! 'c 2 3 4)
(table-insert! 'd 2 3 5)
(table-insert! 'e 2 3 6)
(table-lookup 3 4)
(table-lookup 2 3)
(table-lookup 2 3 4)
(table-lookup 1)
(table-insert! 'x 1 1)
(table-insert! 'y 2 3 4)
(table-lookup 1 1)
(table-lookup 2 3 4)
(table-delete! 2 3 4)
(table-lookup 2 3 4)
;; (table-draw)
```

Your tests should yield:

ok  
a  
ok  
ok  
ok  
ok  
ok  
**#f**  
**#f**  
c  
**#f**  
ok  
ok  
x  
y  
ok  
**#f**

*;I think the answer below by Anonymous is wrong because it's just for the one-dimensional table, the exercise requires for a arbitrary n-dimensional table.*

```

(define (make-table)
  (let ((local-table (list '*table*)))
    (define (assoc key records)
      (cond ((null? records) false)
            ((equal? key (caar records)) (car records))
            (else (assoc key (cdr records)))))

    (define (lookup key-list)
      (define (iter keys table)
        (cond ((null? keys) false) ;为空时
              ((null? (cdr keys)) ;只有一个key时
               (let ((record (assoc (car keys) (cdr table))))
                 (if record
                     (cdr record)
                     false)))
              (else ;有多个key时,先取出一个,用于找到subtable,然后
                   ;剩下的再循环再
                   (let ((subtable (assoc (car keys) (cdr table))))
                     (if subtable
                         (iter (cdr keys) subtable)
                         false))))))

      (iter key-list local-table))

    (define (insert! value key-list)
      (define (iter keys table)
        (cond ((null? table) ;这是当没有找到key对应的subtable时,需要创建新的subtable
               (if (null? (cdr keys))
                   (cons (car keys) value)
                   (list (car keys) (iter (cdr keys) '())))
               ((null? (cdr keys)) ;只有一个key
                (let ((record (assoc (car keys) (cdr table))))
                  (if record
                      (set-cdr! record value)
                      (set-cdr! table
                                (cons (cons (car keys) value)
                                      (cdr table))))))
               (else ;有多个key
                (let ((subtable (assoc (car keys) (cdr table)))))
                  (if subtable

```

```

        (iter (cdr keys) subtable)
        (set-cdr! table
                   (cons (list (car keys)
                               (iter (cdr keys) '())))
                         ;;这里是关键，没
找到subtable时，创建新的，然后循环(cdr keys)
                         (cdr table))))))))
        (iter key-list local-table)
        'ok)
(define (dispatch m)
  (cond ((eq? m 'lookup-proc) lookup)
        ((eq? m 'insert-proc!) insert!)
        (else (error "Unknown operation -- TABLE" m))))
  dispatch))

(define (lookup table . key-list) ((table 'lookup-proc) key-list))
(define (insert! table value . key-list) ((table 'insert-proc!) value key-list))

```

### Anonymous

```

(define (fold-left op init seq)
  (define (iter ans rest)
    (if (null? rest)
        ans
        (iter (op ans (car rest))
              (cdr rest))))
  (iter init seq))

(define (make-table same-key?)
  (define (associate key records)
    (cond ((null? records) #f)
          ((same-key? key (caar records)) (car records))
          (else (associate key (cdr records)))))

  (let ((the-table (list '*table*)))
    (define (lookup keys)
      (define (lookup-record record-list key)
        (if record-list
            (let ((record (assoc key record-list)))
              (if record
                  (cdr record)
                  #f))
            #f))
      (fold-left lookup-record (cdr the-table) keys))

    (define (insert! keys value)
      (define (descend table key)
        (let ((record (assoc key (cdr table))))
          (if record
              record
              (let ((new (cons (list key)
                               (cdr table))))
                (set-cdr! table new)
                (car new)))))
      (set-cdr! (fold-left descend the-table keys)
                value))

    ;; N.B. PRINT will break if a record has a list structure for a value.
    (define (print)
      (define (indent tabs)
        (for-each (lambda (x) (display #\tab)) (iota tabs)))

    (define (print-record rec level)
      (indent level)
      (display (car rec))
      (display ": ")
      (if (list? rec)
          (begin (newline)
                 (print-table rec (1+ level)))
          (begin (display (cdr rec))
                 (newline)))))

    (define (print-table table level)
      (if (null? (cdr table))
          (begin (display "-no entries-")
                 (newline))
          (for-each (lambda (record)
                      (print-record record level))
                    (cdr table)))))

    (print-table the-table 0))

    (define (dispatch m)
      (cond ((eq? m 'lookup) lookup)

```

```

((eq? m 'insert!) insert!)
((eq? m 'print) print)
(else (error "Undefined method" m)))

dispatch))

⇒(define op-table (make-table eq?))
⇒(define put (op-table 'insert!))
⇒(define get (op-table 'lookup))
⇒((op-table 'print))
-no entries-
⇒(put '(letters a) 97) ; Two dimensions
⇒(put '(letters b) 98)
⇒(put '(math +) 43)
⇒(put '(math -) 45)
⇒(put '(math *) 42)
⇒(put '(greek majiscule Λ) 923) ; Three dimensions
⇒(put '(greek miniscule λ) 955)
⇒(put '(min) 42) ; One dimension
⇒(put '(max) 955)
⇒(get '(min))
42
⇒(get '(letters b))
98
⇒(get '(greek majiscule Λ))
923
⇒(get '(dfashoigrar asdfasdf retaqw))
#f
⇒((op-table 'print))
max: 955
min: 42
greek:
    majiscule:
        Λ: 923
    miniscule:
        λ: 955
math:
    +: 43
    *: 42
    -: 45
letters:
    a: 97
    b: 98

```

```
(insert-helper (cdr remaining-keys) new-table)))))  
(insert-helper keys table)  
'ok)
```

@meteorgan - Yes, it will work, but everything will be saved in the same 1D table, which is not the point of the exercise I think

meteorgan

I don't think we need change the procedure make-table. If we use a list as the key, all things can keep the same.

roy-tobin

@meteorgan Most astute. The **code** for this approach is lovely.

sam

Yes, I agree if we just use lists for keys; nothing needs to be changed. However, the exercise intends to implement arbitrary dimension tables (generalizing from 2D).

ada

another solution

```
(define (make-table same-key?)  
  (let ((local-table (list '*table*)))  
  
    (define (assoc key records)  
      (cond ((null? records) #false)  
            ((same-key? key (caar records))  
             (car records))  
            (else (assoc key (cdr records)))))  
  
    (define (lookup keys)  
      (define (iter remain-keys records)  
        (cond  
          ((null? remain-keys) records)  
          ((not (pair? records)) #false)  
          (else (let ((record (assoc (car remain-keys) records)))  
                  (if record  
                      (iter (cdr remain-keys) (cdr record))  
                      #false))))  
        (iter keys (cdr local-table)))  
  
    (define (insert! keys value)  
      (define (iter ks records)  
        (cond  
          ((null? ks) (set-cdr! records value))  
          ((or (null? (cdr records)) (not (pair? (cdr records))))  
           (set-cdr! records (list (cons (car ks) '()) )))  
           (iter (cdr ks) (cadr records)))  
          (else (let ((record (assoc (car ks) (cdr records))))  
                  (if record  
                      (iter (cdr ks) record)  
                      (begin (set-cdr! records  
                               (cons (list (car ks))  
                                     (cdr records)))  
                           (iter (cdr ks) (cadr records)))))))  
        (iter keys local-table))  
  
    (define (dispatch m)  
      (cond  
        ((eq? m 'lookup) lookup)  
        ((eq? m 'insert!) insert!)))  
      dispatch))  
  
    (define (lookup keys table)  
      ((table 'lookup) keys))  
    (define (insert! keys value table)  
      ((table 'insert!) keys value))
```

Yasser Hussain

```
(define (is-table? value)
  (and (pair? value) (eq? (car value) '*table*)))

(define (lookup keys table)
  (cond ((null? keys) (if (is-table? table) #f table))
        ((not (is-table? table)) #f)
        (else (let ((record (assoc (car keys) (cdr table))))
                 (if record
                     (lookup (cdr keys) (cdr record))
                     #f)))))

(define (insert! keys value table)
  (if (null? keys)
      table
      (let ((record (assoc (car keys) (cdr table))))
        (if (= (length keys) 1)
            (if (not record)
                (set-cdr! table (cons (cons (car keys) value) (cdr table)))
                (set-cdr! record value))
            (if (not record)
                (let ((new-table (cons '*table* '())))
                  (set-cdr! table (cons
                                    (cons (car keys) new-table)
                                    (cdr table)))
                  (insert! (cdr keys) value new-table))
                (if (is-table? (cdr record))
                    (insert! (cdr keys) value (cdr record))
                    (let ((new-table (cons '*table* '())))
                      (set-cdr! record new-table)
                      (insert! (cdr keys) value new-table)))))))

(define table (cons '*table* '()))

(insert! (list '+) "add" table)
(insert! (list '-) "sub" table)

(lookup (list '-') table)

(insert! (list '-') "new-sub" table)
(lookup (list '-') table)

(insert! '(c1 c2 c3) 22 table)

(lookup (list '+) table)
(lookup (list 'k1 'k2 'k3) table)
(lookup (list 'k1 'k2 'k3 'k4) table)
(lookup '(c1 c2 c3) table)
```

Anonymous

```
;nested tables
;final values are always wrapped into a table
;so that a cdr of a record is always a table
;this allows to store prefix-equal keys without conflicts
;say, the table handles keys (1 2) and (1 2 3) just fine
;but the code is not very readable imo
(define (empty-table)
  (list 'head))

(define (table-body table)
  (cdr table))

(define (append-table! table key value)
  (let ((newpair (cons key value)))
    (set-cdr! table (cons newpair (table-body table)))))

(define special-key 'xxx)

(define (insert! table keys value)
  (if (null? keys)
      (let ((record (assoc special-key (table-body table))))
        (if (eq? record #f)
            (append-table! table special-key value)
            (set-cdr! record value)))
      (let ((record (assoc (car keys) (table-body table))))
        (cond ((eq? record #f)
```

```

        (append-table! table (car keys) (empty-table))
        (insert! (cdadr table) (cdr keys) value)) ;call insert on the
newly appended sub-table
      (else (insert! (cdr record) (cdr keys) value)))))

(define (lookup table keys)
  (if (null? keys)
      (let ((record (assoc special-key (table-body table))))
        (if (eq? record #f)
            #f
            (cdr record)))
      (let ((record (assoc (car keys) (table-body table))))
        (if (eq? record #f)
            #f
            (lookup (cdr record) (cdr keys))))))

(define t (empty-table))
(insert! t '(1 2) 'a)
(insert! t '(1) 'b)
(lookup t '(1 2))

```

GP

My solution. Simple recursion is enough for this exercise.

```

(define (make-table)
  (let ((local-table (list '*table*)))
    (define (lookup keys)
      (define (iter keys table)
        (let ((subtable (assoc (car keys) (cdr table))))
          (if subtable
              (cond ((null? (cdr keys)) (cdr subtable))
                    (else
                      (iter (cdr keys) subtable)))
              false)))
      (iter keys local-table)))

    (define (gen-new-list keys value)
      (if (null? (cdr keys))
          (cons (car keys) value)
          (list (car keys) (gen-new-list (cdr keys) value)))))

    (define (insert! keys value)
      (define (iter keys table)
        (let ((subtable (assoc (car keys) (cdr table))))
          (if subtable
              (cond ((null? (cdr keys))
                      (set-cdr! subtable value))
                    (else
                      (iter (cdr keys) subtable)))
              (set-cdr! table (cons (gen-new-list keys value) (cdr table))))))
        'ok))
      (iter keys local-table)))

    (define (dispatch m)
      (cond ((eq? m 'lookup-proc) lookup)
            ((eq? m 'insert-proc!) insert!)
            ((eq? m 'display) (display local-table))
            (else (error "Unknown operation -- TABLE" m))))
    dispatch))

```

davl

My make-it-as-concise-as-possible version.

```

(define (make-table)
  (let ((local-table (list '*table*)))
    (define (lookup subtable key-list)
      (cond ((not subtable) #f)
            ((null? key-list) (if (list? subtable) subtable (cdr subtable)))
            (else (lookup (assoc (car key-list) (cdr subtable)) (cdr key-list)))))

    (define (insert! subtable key-list value)
      (if (null? key-list)
          (set-cdr! subtable value)
          (let ((cur-key (car key-list))
                (rest-keys (cdr key-list)))
            (subtable-rest (if (list? subtable) (cdr subtable) '())))
            (let ((record (assoc cur-key subtable-rest)))
              (if (not record)
                  (begin
                    (set! record (list cur-key))

```

```

(set-cdr! subtable (cons record subtable-rest)))
(insert! record rest-keys value)))))

(define (dispatch m)
  (cond ((eq? m 'lookup-proc) (lambda (key-list) (lookup local-table key-
list)))
        ((eq? m 'insert-proc!) (lambda (key-list value) (insert! local-table
key-list value)))
        (else (error "Unknown operation -- TABLE" m))))
  dispatch))

```

Sphinxsky

```

(define (make-table same-key?)

(define (new-table name) (list name))

(define (table-add! table new-record)
  (set-cdr! table (cons new-record (cdr table)))
  table)

(define (table-add-rec! table keys-list value)
  (cond ((null? keys-list)
         (set-cdr! table value)
         table)
        (else
         (table-add!
          table
          (table-add-rec!
           (new-table (car keys-list))
           (cdr keys-list)
           value)))))

(define (assoc- key records)
  (cond ((null? records) #f)
        ((same-key? key (caar records))
         (car records))
        (else
         (assoc- key (cdr records)))))

(let ((local-table (new-table '*table*)))

(define (lookup keys-list)
  (define (rec now-table now-keys)
    (if (null? now-keys)
        (cdr now-table)
        (let ((subtable (assoc- (car now-keys) (cdr now-table))))
          (if subtable
              (rec subtable (cdr now-keys))
              #f))))
    )

(let ((len (length keys-list)))
  (let ((subtable (assoc- len (cdr local-table))))
    (if subtable
        (rec subtable keys-list)
        #f)))

(define (insert! keys-list value)
  (define (rec now-table now-keys)
    (if (null? now-keys)
        (set-cdr! now-table value)
        (let ((subtable (assoc- (car now-keys) (cdr now-table))))
          (if subtable
              (rec subtable (cdr now-keys))
              (table-add-rec! now-table now-keys value)))))

(let ((len (length keys-list)))
  (let ((subtable (assoc- len (cdr local-table))))
    (if subtable
        (rec subtable keys-list)
        (table-add!
         local-table
         )))))

```

```

        (table-add-rec!
         (new-table len)
         keys-list value)))))

'done)

(define (dispatch m)
  (cond ((eq? m 'lookup-proc) lookup)
        ((eq? m 'insert-proc!) insert!)
        (else (error "Unknown operation -- TABLE" m)))))

(dispatch)

(define operation-table (make-table equal?))
(define get (operation-table 'lookup-proc))
(define put (operation-table 'insert-proc!))

```

antbbn

Maybe not exactly in the spirit of the exercise, but I noticed that changing the "Two dimensional table" structure to just accomodate for nested one dimensional tables (i.e. an extra (cons \*table ..) after the key) makes the code very straightforward

```

; insert! and lookup from the one dimensional table implementation

(define (table? t)
  (and (pair? t) (eq? '*table* (car t)))))

(define (lookup-generic table key . rest-of-keys)
  (let ((subtable-or-record (lookup key table)))
    (cond ((not subtable-or-record) false)
          ((null? rest-of-keys) subtable-or-record)
          ((table? subtable-or-record) (apply lookup-generic subtable-or-record rest-of-keys))
          (else (error "LOOKUP-GENERIC key is not a subtable" key subtable-or-record)))))

(define (insert-generic! table value key . rest-of-keys)
  (if (null? rest-of-keys) ; on the last key
      (insert! key value table)
      (let ((subtable-or-record (lookup key table)))
        (if (table? subtable-or-record)
            (apply insert-generic! subtable-or-record value rest-of-keys)
            (let ((new-subtable (make-table)))
              (insert! key new-subtable table)
              (apply insert-generic! new-subtable value rest-of-keys))))))


```

tch

Actually, your idea is so brilliant, it reuses existing code maximally, and distinguishes a subtable with a record.

denis manikhin

```

#lang sicp
(define (make-table same-key?)
  (let ((local-table (list '*table*)))

    (define (s-assoc key-x? key records)
      (cond ((null? records) false)
            ((not (key-x? records)) (s-assoc key-x? key (cdr records)))
            ((same-key? key (caar records)) (car records))
            (else (s-assoc key-x? key (cdr records)))))

    (define (key-table? record)
      (pair? (cdr record)))

    (define (key-value? record)
      (not (pair? (cdr record)))))

    (define (rec-lookup i-table key rest-of-keys)
      (let ((subtable (s-assoc key-table? key (cdr i-table))))
        (cond ((null? rest-of-keys) (lookup i-table key))
              (subtable (rec-lookup subtable (car rest-of-keys) (cdr rest-of-keys)))
              (else false)))))


```

```

(define (lookup i-table key)
  (let ((record (s-assoc key-value? key (cdr i-table))))
    (if record
        (cdr record)
        false)))

(define (rec-insert! i-table value key rest-of-keys )
  (let ((subtable (s-assoc key-table? key (cdr i-table))))
    (cond ((null? rest-of-keys) (insert! i-table key value))
          (subtable (rec-insert! (cdr subtable)
                                value
                                (car rest-of-keys)
                                (cdr rest-of-keys)))
          (else (begin (set-cdr! i-table
                                  (cons (list key)
                                         (cdr i-table)))
                     (rec-insert! (cadr i-table)
                                 value
                                 (car rest-of-keys)
                                 (cdr rest-of-keys))))))
    'ok))

(define (insert! i-table key value)
  (let ((record (s-assoc key-value? key (cdr i-table))))
    (if record
        (set-cdr! record value)
        (set-cdr! i-table (cons (cons key value)
                               (cdr i-table)))))
  'ok)

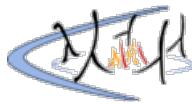
(define (dispatch m)
  (cond ((eq? m 'lookup-proc)
         (lambda (key . key-list) (rec-lookup local-table
                                              key key-list)))
        ((eq? m 'insert-proc!)
         (lambda (value key . key-list) (rec-insert! local-table
                                                       value
                                                       key
                                                       key-list)))
        (else (error "Unknown operation: TABLE" m))))
  dispatch)

(define (predicat? test-key tbl-key)
  (= test-key tbl-key))

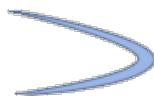
(define tbl (make-table predicat?))

(((tbl 'insert-proc!) 155 21)
 (((tbl 'lookup-proc) 21)
  ((tbl 'insert-proc!) 255 21 21)
  (((tbl 'lookup-proc) 21 21)
   ((tbl 'lookup-proc) 21)
   (((tbl 'insert-proc!) 355 21)
    (((tbl 'lookup-proc) 21 21)
     ((tbl 'insert-proc!) 355 21)
     (((tbl 'lookup-proc) 21)
      ((tbl 'lookup-proc) 22)
      (((tbl 'lookup-proc) 22 22)
       ((tbl 'insert-proc!) 455 22 22)
       (((tbl 'lookup-proc) 22 22)
        ((tbl 'insert-proc!) 555 22 22)
        (((tbl 'lookup-proc) 22 22)

```



# sicp-ex-3.26



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.25) | Index | Next exercise (3.27) >>

flamingo

```
; node == ( ( pair key value ) left-ptr right-ptr )

(define (entry tree) (car tree))
(define (left-branch tree) (cadr tree))
(define (right-branch tree) (caddr tree))

(define (make-tree entry left right)
  (list entry left right))

(define (adjoin-set x set)
  (cond ((null? set) (make-tree x '() '()))
        ((= (car x) (car (entry set))) set)
        ((< (car x) (car (entry set)))
         (make-tree (entry set)
                    (adjoin-set x (left-branch set))
                    (right-branch set)))
        ((> (car x) (car (entry set)))
         (make-tree (entry set)
                    (left-branch set)
                    (adjoin-set x (right-branch set)))))

(define (make-table)
  (let ((local-table '()))

    (define (lookup key records)
      (cond ((null? records) #f)
            ((= key (car (entry records))) (entry records))
            ((< key (car (entry records))) (lookup key (left-branch records)))
            ((> key (car (entry records))) (lookup key (right-branch records)))))

    (define (insert! key value)
      (let ((record (lookup key local-table)))
        (if record
            (set-cdr! record value)
            (set! local-table (adjoin-set (cons key value) local-table)))))

    (define (get key)
      (lookup key local-table))

    (define (dispatch m)
      (cond ((eq? m 'get-proc) get)
            ((eq? m 'insert-proc) insert!)
            ((eq? m 'print) local-table)
            (else (error "Undefined operation -- TABLE" m))))
      dispatch))

  (define table (make-table))
  (define get (table 'get-proc))
  (define put (table 'insert-proc))
```

```
⇒(put 43 'a)
⇒(put 42 'b)
⇒(put 41 'c)
⇒(put 67 'z)
⇒(put 88 'e)

⇒(table 'print)
((43 . a)
 ((42 . b) ((41 . c) () ()))
 ((67 . z) () ((88 . e) () ()))

⇒(get 88)
(88 . e)
```

gws

Solution for multi-dimensional tables

```

; helper methods

(define (make-record key value)
  (list (cons key value) nil nil))
(define (get-key record) (caar record))
(define (get-value record) (cdar record))
(define (set-key! record new-key) (set-car! (car record) new-key))
(define (set-value! record new-value) (set-cdr! (car record) new-value))
(define (get-left record) (cadr record))
(define (get-right record) (caddr record))
(define (set-left! record new-left) (set-car! (cdr record) new-left))
(define (set-right! record new-right) (set-car! (cddr record) new-right))

(define (assoc key records)
  (cond ((null? records) false)
        ((equal? key (get-key records)) (get-value records))
        ((< key (get-key records)) (assoc key (get-left records)))
        (else (assoc key (get-right records)))))

(define (add-record key value table)
  (define (iter record parent set-action)
    (cond ((null? record) (let ((new (make-record key value)))
                           (set-action parent new)
                           (car new)))
          ((equal? key (get-key record)) (set-value! record value)
           (car record))
          ((< key (get-key record)) (iter (get-left record) record set-left!))
          (else (iter (get-right record) record set-right!))))
    (iter (cdr table) table set-cdr!)))

; the procedure

(define (make-table)

(let ((local-table (list '*table*)))

  (define (lookup keys)
    (define (iter keys records)
      (if (null? keys) records
          (let ((found (assoc (car keys) records)))
            (if found (iter (cdr keys) found)
                false))))
    (iter keys (cdr local-table)))

  (define (insert! keys value)
    (define (iter keys subtable)
      (cond ((null? (cdr keys)) (add-record (car keys) value subtable))
            (else (let ((new (add-record (car keys) nil subtable)))
                    (iter (cdr keys) new)))))
    (iter keys local-table)
    'ok)

  (define (print) (display local-table) (newline))

  (define (dispatch m)
    (cond ((eq? m 'lookup-proc) lookup)
          ((eq? m 'insert-proc!) insert!)
          ((eq? m 'print) print)
          (else (error "Unknown operation - TABLE" m)))
    dispatch))

  (define operation-table (make-table))
  (define get (operation-table 'lookup-proc))
  (define put (operation-table 'insert-proc!))
  (define print-table (operation-table 'print)))

```

sam

### Solution using mutable trees

```

(define (entry tree) (car tree))
(define (left-branch tree) (caar tree))
(define (right-branch tree) (caddr tree))
(define (make-tree entry-record left right)
  (list entry-record left right))

(define (adjoin-set x set)
  (cond ((null? set) (make-tree x '() '()))
        ((< (car x) (car (entry set))))
        (let ((left (left-branch set))
              (set! left (adjoin-set x left))))
        ((> (car x) (car (entry set))))
        (let ((right (right-branch set))
              (set right (adjoin-set x right))))))

```

```

(define (assoc key record-tree)
  (cond ((null? record-tree) false)
        ((eq? key (car (entry record-tree)) (entry record-tree)))
        ((< key (car (entry record-tree))) (assoc key (left-branch record-tree)))
        ((> key (car (entry record-tree))) (assoc key (right-branch record-tree))))))

(define (make-table)
  (let (local-table (list '*table*))
    (define (look-up key)
      (let ((record (assoc key (cdr local-table))))
        (if record
            (cdr record)
            false)))

    (define (insert! key value)
      (let ((record (assoc key (cdr local-table))))
        (if record
            (set-cdr! record value)
            (adjoin-set (cons key value) (cdr local-table)))
        'ok)))

    (define (dispatch m)
      (cond ((eq? m 'lookup-proc) look-up)
            ((eq? m 'insert-proc!) insert!)
            (else "No such operation on table" m)))
    dispatch)))

```

GP

I like the above solution from sam with internal "assoc", because it hides the common logic of two separate steps (lookup and insert) using procedural abstraction, which is what we have learnt in Chap 1. However it is still not ideal, as it requires an additional definition for "adjoin-set", which repeats some similar logic under "assoc". In fact, if a key cannot be found in a tree, it needs to traverse the tree twice to insert the value. The first round only checks if the key is in the tree. Then it goes over the tree again to insert it in the proper position, if the key dose not existed.

I am wondering if there is an even higher level of abstraction that encapsulates the similar code between "assoc" and "adjoin-set".

My solution dose not go along this direction. I simply let the similar code be naked and presented in lookup and insert!. But I tried to write the two parts of the code in a symmetrical way that have a similar structure. With this way, we gain some readability even without procedural abstraction. For small program like this, it works fine.

In addition, I create one more abstraction layer of node, which improves a bit readability and provides some flexibility for different ways of tree implementation.

```

(define (make-table)
  (define (entry tree) (car tree))
  (define (left-branch tree) (cadr tree))
  (define (right-branch tree) (caddr tree))
  (define (make-tree entry left right)
    (list entry left right))

  (define (node-key entry)
    (car entry))
  (define (node-value entry)
    (cdr entry))
  (define (make-node key value)
    (cons key value))
  (define (set-value! node value)
    (set-cdr! node value))

  (let ((root (list )))

    (define (lookup key)
      (define (iter tree)
        (cond ((null? tree) false)
              (else
                (let ((node (entry tree))
                      (left (left-branch tree))
                      (right (right-branch tree)))
                  (cond ((= key (node-key node)) (node-value node))
                        ((< key (node-key node)) (iter left))
                        ((> key (node-key node)) (iter right)))))))

```

```

(define (insert! key value)
  (define (iter tree)
    (cond ((null? tree) (make-tree (make-node key value) '() '())
           (else
             (let ((node (entry tree)))
               (left (left-branch tree))
               (right (right-branch tree)))
             (cond ((= key (node-key node))
                   (set-value! node value)
                   tree)
                   ((< key (node-key node))
                    (make-tree node (iter left) right))
                   ((> key (node-key node))
                    (make-tree node left (iter right))))))
           (set! root (iter root)))))

(define (dispatch m)
  (cond ((eq? m 'lookup-proc) lookup)
        ((eq? m 'insert-proc!) insert!)
        ((eq? m 'display) (display root) (newline)))
        (else (error "Unknown operation -- TREE" m)))))

(dispatch))

(define (show tree) (tree 'display))
(define (lookup tree key) ((tree 'lookup-proc) key))
(define (insert! tree key value) ((tree 'insert-proc!) key value))

;; TESTING
(define t (make-table))
(show t)
(insert! t 7 'a)
(show t)
(insert! t 3 'b)
(show t)
(insert! t 9 'c)
(show t)
(insert! t 5 'd)
(show t)
(insert! t 1 'e)
(show t)
(insert! t 11 'f)
(show t)
(lookup t 5)
(lookup t 1)
(lookup t 9)
(insert! t 9 'xxx)
(lookup t 9)
(lookup t 27)

```

joe w

joe w This works with nested trees, meaning the value of an entry in a tree can either be a single value or another binary tree that can be searched via the lookup procedure. You add nested trees by providing more than one key to the list of keys for an entry using the insert procedure!

**NOTE:** You will see a lot of debugging data if you run this without removing the display statements.

```
#lang sicp

(define (new-table)
  (define (make-entry key value) (cons key value))
  (define (entry tree) (caar tree))
  (define (left-branch tree) (cadr tree))
  (define (right-branch tree) (caddr tree))
  (define (make-tree entry left right)
    (list entry left right))
  (define (make-tree-node x) (make-tree x '() '())))

  (define (set-left-branch! tree entry) (set-car! (cdr tree) entry))
  (define (set-right-branch! tree entry) (set-car! (cddr tree) entry))

  (let ((table (list '*table*)))

    (define (lookup keys)
      (let ((record (assoc keys (cdr table))))
        (if record
            (cdr record)
            false)))

    (define (assoc keys records)
      (define (handle-matches keys record)
        (cond ((null? keys) record)
              ...)))
```

```

        ((pair? (cdr record)) (assoc keys (cdr record)))
        (else #f)))
(display (list "assoc params" "KEYS:" keys "RECORDS:" records)) (newline)
(cond ((or (null? records) (null? (car records))) false)
      ((equal? (car keys) (caar records))
       (handle-matches (cdr keys) (car records)))
       ((< (car keys) (caar records))
        (assoc keys (left-branch records)))
       (else (assoc keys (right-branch records)))))

(define (nest-trees-for-remaining-keys keys value)
  (display (list "null-set-handler params" "keys:" keys "value:" value))
  (newline)
  (if (null? keys) value
      (make-tree-node (make-entry (car keys) (nest-trees-for-remaining-keys (cdr
keys) value))))))

;(keys) -> atom b-> atom b|table a b-> atom b|table a b
(define (match-handler keys value entry)
  (display (list "match-handler params" "keys:" keys "entry:" entry))
  (newline)
  (cond ((null? keys) value)
        ((pair? entry) (adjoin-set! keys value entry))
        (else (nest-trees-for-remaining-keys keys value)))))

(define (adjoin-set! keys value set)
  (display (list "adjoin-set:" set))
  (newline)
  (let ((new-key (car keys)))
    (cond ((null? set) (nest-trees-for-remaining-keys keys value))
          ((eq? new-key (caar set))
           (set-cdr! (car set) (match-handler (cdr keys) value (cdar set)))
           set)
          ((< new-key (entry set))
           (set-left-branch! set (adjoin-set! keys value (left-branch set)))
           set)
          ((> new-key (entry set))
           (set-right-branch! set (adjoin-set! keys value (right-branch set)))
           set)))))

(define (insert-tree! keys value)
  (display (list "INSERT TREE PARAMS:" keys value table))
  (set-cdr! table (adjoin-set! keys value (cdr table)))
  (display table)
  'ok)

(define (print) (display table) (newline))
(define (dispatch m)
  (cond ((eq? m 'lookup) (lambda (keys) (lookup keys)))
        ((eq? m 'print) print)
        ((eq? m 'insert) (lambda (keys value) (insert-tree! keys value)))
        (else "Invalid command")))
  dispatch))

;PROCEDURAL INTERFACES
(define t4 (new-table))
(define (insert! table keys value)
  ((table 'insert) keys value))
(define (print table)
  ((table 'print)))
(define (lookup table keys)
  ((table 'lookup) keys))

;TESTS
(insert! t4 '(76 -456) 'jesuit)
(insert! t4 '(76 -834) 'chomsky)
(insert! t4 '(76 -1000) 'regime)
(insert! t4 '(50 1/2) 'francoi)
(insert! t4 '(50 1/2 .333) 'twei)
(insert! t4 '(50 1/2 .666) 'cambodia)
(lookup t4 '(50 1/2 .333)) ;twei
(lookup t4 '(76 -456)) ;false because it should have been overwritten
(insert! t4 '(76 -456) 'carmelite)
(print t4);(*table* (76 (-456 . carmelite) ((-834 . chomsky) ((-1000 . regime) () ()) ())
()) ((50 (1/2 (0.333 . twei) () ((0.666 . cambodia) () ()) () () () ()))))
```

roy-tobin

@joew I find subtle trouble via a **torture test** upon two insertions:

```

antix21:~/src/scheme/solutions2 % racket -i -p neil/sicp
Welcome to Racket v7.9 [bc].
> (load "joe")
> (define t1 (make-table))
> (insert! t1 (list 379) '(1 . 2))
'ok
> (insert! t1 (list 379 666) 1.618)
; mcar: contract violation
```

```

;   expected: mpair?
;   given: 1
; [,bt for context]

```

x3v

forgot about the existence of assoc when doing this... fun exercise nonetheless

```

(define (binary-tree-table)
  (let ((head '()))
    (define (make-node key value)
      (list key value '() '()) ; (key value left right))
    (define (get-key node) (car node))
    (define (get-value node) (cadr node))
    (define (left node) (caddr node))
    (define (right node) (cadddr node))
    (define (set-left! node ptr) (set-car! (cddr node) ptr))
    (define (set-right! node ptr) (set-car! (cdddr node) ptr))
    (define (set-value! node val) (set-car! (cdr node) val))
    (define (leaf? node)
      (and (null? (left node)) (null? (right node))))
    (define (lookup key) ;; returns value if key in tree else #f
      (define (iter node)
        (cond ((null? node) #f)
              ((eq? key (get-key node)) (get-value node))
              ((leaf? node) #f)
              (else (iter (if (> key (get-key node))
                             (right node)
                             (left node)))))))
      (iter head))
    (define (insert key value)
      (let ((new-node (make-node key value)))
        (define (iter node)
          (cond ((= key (get-key node)) (set-value! node value))
                ((> key (get-key node))
                 (if (null? (right node))
                     (set-right! node new-node)
                     (iter (right node))))
                (else (if (null? (left node))
                           (set-left! node new-node)
                           (iter (left node))))))
        (if (null? head)
            (set! head new-node)
            (iter head))))
    (define (dispatch m)
      (cond ((eq? m 'insert) insert)
            ((eq? m 'lookup) lookup)
            ((eq? m 'head) head)
            (else (error "incorrect usage" m))))
    dispatch))

(define table (binary-tree-table))
(define (insert table key value)
  ((table 'insert) key value))
(define (lookup table key)
  ((table 'lookup) key))

;; tests
(insert table 5 'y)
(insert table 3 'o)
(insert table 8 'a)
(insert table 4 'b)
(table 'head) ; (5 y (3 o () (4 b () ())) (8 a () ()))
(lookup table 4) ; b
(lookup table 5) ; y
(lookup data 6) ; #f

```

roy-tobin

Nothing profound here. One thing new is that the function to compare keys is a formal parameter, facilitating "keys that can be ordered in some way (e.g. alphabetically)." A testbench and analysis of all solutions above is [here](#).

```

(define (make-table cmpfunc)
  (define root-tree '())
  (define st-uninitialized #f) ; the result returned for unsuccessful lookup
  (define (st-beget k v) (list k '() '() v))
  (define (st-get-key st) (car st)) ; 1st element
  (define (st-lefthead st) (cdr st)) ; 2nd element
  (define (st-righthead st) (cddr st)) ; 3rd element

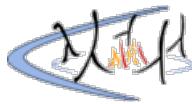
```

```

(define (st-get-value st) (caddr st)) ; 4th element
(define (st-leftsubtree st) (car (st-lefthead st)))
(define (st-rightsubtree st) (car (st-righthead st)))
(define (st-set-value st val) (set-car! (cdddr st) val))

(define (st-graft head k v)
  (if (null? (car head))
      (set-car! head (st-beget k v))
      (st-insert (car head) k v))
  )
(define (st-insert st key val) ; contract is "st (the subtree) is always non-null"
  (let ((result (cmpfunc (st-get-key st) key)))
    (cond ((< result 0) (st-graft (st-righthead st) key val))
          ((> result 0) (st-graft (st-lefthead st) key val))
          (else (st-set-value st val))))
  )
(define (st-lookup st key) ; contract is "st may or may not be null"
  (if (null? st)
      st-uninitialized
      (let ((result (cmpfunc (st-get-key st) key)))
        (cond ((< result 0) (st-lookup (st-rightsubtree st) key))
              ((> result 0) (st-lookup (st-leftsubtree st) key))
              (else (st-get-value st)))))
  )
(define (st-root-insert key val)
  (if (null? root-tree)
      (set! root-tree (st-beget key val))
      (st-insert root-tree key val))
  )
(define (dispatch m)
  (cond ((eq? m 'insert) st-root-insert)
        ((eq? m 'lookup) (lambda (key) (st-lookup root-tree key)))
        ((eq? m 'dump) root-tree) ; debugging
        (else (error "Error: unknown dispatch message -- btable" m)))
  )
  dispatch
)
(define (insert! bt key val) ((bt 'insert) key val))
(define (lookup bt key) ((bt 'lookup) key))

```



# sicp-ex-3.27

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.26) | Index | Next exercise (3.28) >>

meteorgan

The (memo-fib) never computes the same result twice. the call tree is a linear list. If define memo-fib to be (memorize fib), the procedure won't work. because fib calls itself to compute (fib n) before applies memorize.

sam

```
(define memo-fib (memoize fib))
```

will not work because when evaluating code for (memoize fib); (fib x) will be evaluated whose enclosing environment is the global environment and thus memoization is lost.

Manu Halvagai

```
(define memo-fib (memoize fib))
```

will actually work but not very well. For example, the call (memo-fib 3) would memoize fib(3), but not fib(1) or fib(2) because these are recursive calls to the unmemoized version of the function. The proper implementation memoizes even the recursive calls.

John-Gaines

Instead of recoding fib into memo-fib as shown in SICP (which seems like a very error-prone way of adding memoize to an existing function). The following works in DrRacket using #lang racket

```
(set! fib (memoize fib))
```

It sets fib in the global environment to (memoize fib) which also causes the recursive calls within fib to call the redefined version.

sam

This will not work under the environment model for evaluation as described in the book. This is because f in memoize will be bound to procedure object that was created via lambda expression for fib; and this procedure object still has global environment as the enclosing environment. (note, all procedures can be created only via lambda expressions).

ericwen229

This WILL work as the fib in global environment HAS CHANGED. When f in memoize gets invoked and searches for fib it will find the redefined version of fib (in global environment as its enclosing environment), which is decorated by memoize.

Yura Rubon

The number of steps is not proportional to n. It is O(n^2) due to assoc procedure. This is confirmed experimentally. The order of growth is very similar to quadratic. With an increase in n by 10 times, the calculation time increased by about 100 times. You can easily check it yourself.



# sicp-ex-3.28

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

<< Previous exercise (3.27) | Index | Next exercise (3.29) >>

---

meteorgan

```
(define (or-gate a1 a2 output)
  (define (or-action-procedure)
    (let ((new-value
           (logic-or (get-singal a1) (get-signal a2))))
      (after-delay or-gate-delay
        (lambda ()
          (set-signal! output new-value)))))

  (add-action! a1 or-action-procedure)
  (add-action! a2 or-action-procedure))

(define (logic-or s1 s2)
  (if (or (= s1 1) (= s2 1))
      1
      0))
```

---

Last modified : 2012-05-05 03:12:36  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-3.29

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.28) | Index | Next exercise (3.30) >>

genovia

`; ;or-gate implementation : https://en.wikipedia.org/wiki/OR_gate`

```
(define (or-gate a1 a2 output)
  (let ((c (make-wire))
        (d (make-wire)) (e (make-wire))
        (f (make-wire)) (g (make-wire)))
    (and-gate a1 a1 d)
    (and-gate a2 a2 e)
    (inverter d f)
    (inverter e g)
    (and-gate f g c)
    (inverter c output)
  'ok))
```

meteorgan

`; ; a  
;; (A or B) is equivalent to (not ((not A) and (not B)))`

```
(define (or-gate a1 a2 output)
  (let ((c1 (make-wire))
        (c2 (make-wire))
        (c3 (make-wire)))
    (inverter a1 c1)
    (inverter a2 c2)
    (and-gate c1 c2 c3)
    (inverter c3 output)))
```

`; ; b  
the delay is the sum of and-gate-delay plus twice inverter-delay.`

Sphinxsky

the delay is the sum of and-gate-delay plus triple inverter-delay.

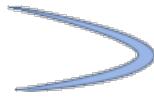
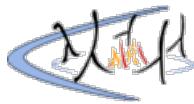
dzy

twice. because only one of a1,a2 changes so that inverter only act once.

squarebat

Even if both inputs were to change simultaneously, the delay would still be twice inverter + and delay, since both inverters for the input wires work at the same time.

# sicp-ex-3.30



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.29) | Index | Next exercise (3.31) >>

3pmtea

```
;;
(define (ripple-carry-adder a-list b-list s-list c)
  (let ((c-list (map (lambda (x) (make-wire)) (cdr a-list)))
        (c-0 (make-wire)))
    (map full-adder
          a-list
          b-list
          (append c-list (list c-0)))
    s-list
    (cons c c-list)))
'ok))
```

genovia

```
;;a
(define (ripple-carry-adder Ak Bk Sk C)
  (define (iter A B S c-in c-out)
    (if (null? A)
        S
        (begin (full-adder (car A) (car B)
                           c-in (car S) c-out)
               (iter (cdr A) (cdr B) (cdr S)
                     (c-out) (make-wire))))))
  (iter Ak Bk Sk C (make-wire)))
```

meteorgan

```
;;a
(define (ripple-carry-adder a b s c)
  (let ((c-in (make-wire)))
    (if (null? (cdr a))
        (set-signal! c-in 0)
        (ripple-carry-adder (cdr a) (cdr b) (cdr s) c-in))
    (full-adder (car a) (car b) c-in (car s) c)))
```

Anonymous

```
(define (ripple-carry-adder a b c-in sum)
  (if (not (null? a))
      (let ((carry (make-wire)))
        (full-adder (car a) (car b) c-in (car sum)
                    carry)
        (ripple-carry-adder (cdr a) (cdr b) carry (cdr sum)))))

  ;;some convinience methods to test the adder
  (define (build-wires input-signals)
    (if (null? input-signals)
        '()
        (let ((new-wire (make-wire)))
          (set-signal! new-wire (car input-signals))
          (cons new-wire (build-wires (cdr input-signals))))))

  (define (get-signals wires)
    (map (lambda (w) (get-signal w)) wires))

  ;;biggest digit first
  (define (to-binary number)
    (if (< number 2)
        (list number)
        (cons (mod number 2) (to-binary (div number 2)))))

  ;;simulation
```

```

(define the-agenda (make-agenda))
(define inverter-delay 1)
(define and-gate-delay 1)
(define or-gate-delay 1)

(define a (build-wires '(0 0 1 0 0 0))) ;;4
(define b (build-wires '(1 1 0 1 0 0))) ;;11
(define s (build-wires '(0 0 0 0 0 0)))
(define c-in (make-wire))
(ripple-carry-adder a b c-in s)

(propagate)
(get-signals s) ;;should be 15

```

Added a couple of convenience procedures if someone wants to play with adders and multipliers

LinYihai

the delays of An or Bn are always 0, so the delays of Cn and Sn are only depends on Cn-1-delay. the recursion formula of Cn-delay is

```

(+ Cn-1-delay
  (* 2 and-gate-delay)
  or-gate-delay
  (max or-gate-delay
       (+ and-gate-delay
          inverter-delay)))

```

and C0-delay equals 0, so the general formula of Cn-delay is

```

(+ (* 2 n and-gate-delay)
   (* n or-gate-delay)
   (* n (max or-gate-delay
             (+ and-gate-delay
                inverter-delay))))

```

Sn-delay is

```

(+ Cn-1-delay
  (* 2
     and-gate-delay)
  (* 2
     (max or-gate-delay
          (+ and-gate-delay
             inverter-delay))))

```

equals

```

(+ (* 2 n and-gate-delay)
   (* (- n 1) or-gate-delay)
   (* (+ n 1) (max or-gate-delay
                     (+ and-gate-delay
                        inverter-delay))))

```

Sphinxsky

```

(define (ripple-carry-adder a-list b-list s-list c)
  (if (null? a-list)
      (begin (set-signal! c 0) c)
      (begin
        (full-adder
          (car a-list)
          (car b-list)
          (ripple-carry-adder
            (cdr a-list)
            (cdr b-list)
            (cdr s-list)
            (make-wire)))
        (car s-list)
        c)))

```

```

c) )))

;; =====test=====

(define (inc init)
  (lambda (m)
    (cond ((number? m)
           (set! init (+ init m)))
          ((eq? m 'get) init)))

(define counter (inc 0))

(define (read-count)
  (counter 1)
  (counter 'get))

(define (make-wire)
  (cons 'c (read-count)))

(define set-signal! set-cdr!)

(define (full-adder a b c-in sum c-out)
  (newline)
  (display (list a b c-in sum c-out)))

(load "ripple-carry-adder.scm")

(ripple-carry-adder
  (list 1 2 3 4 5 6 7 8)
  (list 1 2 3 4 5 6 7 8)
  (list 1 2 3 4 5 6 7 8)
  (cons 'c 'out))

; (define t-and and-gate-delay)
; (define t-or or-gate-delay)
; (define t-not inverter-delay)

; (define t-ha-s (+ t-and (max t-or (+ t-not t-and))))
; (define t-ha-s
;   ; (if (>= t-or (+ t-not t-and))
;   ;     (+ t-and t-or)
;   ;     (+ (* 2 t-and) t-not)))

; (define t-ha-c t-and)

; (define t-fa-s (+ t-ha-s (max 0 t-ha-s)))
; (define t-fa-s (* 2 t-ha-s))

; (define t-fa-c (+ t-or (max t-ha-c (+ t-ha-c (max 0 t-ha-s)))))
; (define t-fa-c (+ t-or t-ha-c t-ha-s))
; (define t-fa-c (+ t-or t-and t-ha-s))

; (= (- t-fa-s t-fa-c) (- t-ha-s t-or t-and))
; (if (>= t-or (+ t-not t-and))
;     (= t-fa-s t-fa-c)
;     (>= t-fa-s t-fa-c))
; (= (max t-fa-s t-fa-c) t-fa-s)
; (define t-fa t-fa-s)

; (define t-rca-c (+ t-fa-c (max 0 0 t-rca-c1)))
; (define t-rca-c (+ t-fa-c t-rca-c1))
; (define t-rca-ci (+ t-fa-c t-rca-ci+1))

; (define t-rca-s1 (+ t-fa-s (max 0 0 t-rca-c1)))
; (define t-rca-s1 (+ t-fa-s t-rca-c1))

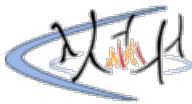
; (define t-rca (max t-rca-c t-rca-s1 t-rca-s2 ... t-rca-sn))
; (define t-rca t-rca-s1)
; (define t-rca (+ t-fa-s t-rca-cn (* (- n 1) t-fa-c)))

; (define t-rca-cn 0)

; (define t-rca
;   ; (+ (* (- n 1) t-or)
;   ;     (* (- n 1) t-and)
;   ;     (* (+ n 1) t-ha-s)))
; (define t-rca
;   ; (if (>= t-or (+ t-not t-and))
;   ;     (* 2 n (+ t-or t-and))
;   ;     (+ (* (- n 1) t-or)
;   ;         (* (+ n 1) t-not)))

```

```
; (* (+ (* 3 n) 1) t-and))))
```



# sicp-ex-3.31

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.30) | Index | Next exercise (3.32) >>

For our event-driven system, it needs an change in bit of a wire to trigger an action (invert, or, add, etc). However, when all the wires are initialized to 0, no actions will be triggered. Therefore we need to trigger those actions manually during the initialization stage.

sam

we use add-action in our function element definitions. If we don't initialize; the time when this function element is inserted in the system will not be recorded on the agenda. Then it is possible, when agenda is simulated via propagate, the function element will not be simulated at all.

joe w

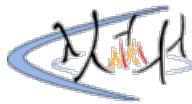
In the specific example, calling propagate after changing the first signal to 1 without running all the actions won't result in the inverter bit having already been flipped and the and gate will fail.

Rather Iffy

At any moment, not necessarily moment 0, when an action procedure is added to a wire that wire and the wires it is connected to by a function box can hold signal values that are not in the correct relationship as defined by that function box. Only when the signal value on the wire changes will the correct relationship between inputs and the output after a delay be enforced. To ensure correctness from the earlier moment of adding the function box it is necessary that the action procedure is run immediately.

thomas

I think the explanations given above are a little hazy. So here's my take: When we build a function (eg define an and-gate) we call add-action! to the input wires. This procedure adds the appropriate procedure (add-action-procedure) to the input-wires, which will be called when the wire signal changes. When that happens it will add commands to the appropriate time-slot in the agenda, that will then change signals and possibly build new-functions. However this will not happen when we run propagate (without an initialization), since the propagate-procedure will only call the procedures stored in the agenda, but the commands will only get stored in the agenda when the procedures are run in the first place. So in order to store them in the agenda we'll have to run the procedure when it gets added to the wire. When we then run propagate, the first (initilaizer) commands can be executed, which change signals and possibly build new-functions-that add procedures to new wires and store new commands in the agenda until we're finished.



# sicp-ex-3.32

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.31) | Index | Next exercise (3.33) >>

```
((lambda () (set-signal! output 0)) (lambda () (set-signal! output 1)))  
  
If the execution order of the above action list changed, the final result of output will be  
different.
```

sam

If two or more input-events trigger an output-event simultaneously; we stop the clock.

And do following,

- 1)change the output-event for input-event1
- 2)change the output-event for input-event2 (note that this new output-event is calculated with new input-event1).
- 3)....and so on.

This has to be done in FIFO order. Otherwise, affecting change for input-event2 will not be correct because output-event would then not reflect the changed input-event1.

CrazyAlvaro

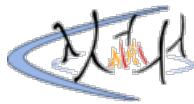
For and-gate: both a1 a2 have and-action-procedure which contains after-delay

the point here is that new-value is calculated before delay  
if the segments' list has LIFO, and the first wire a1 change first, then the result will  
be incorrect  
(a1, a2)  
(0, 1) initial  
(1, 1) a1 0 -> 1, then a1's and-action-procedure called, a1's new-value = 1, lambda  
procedure inserted into segment  
(1, 0) a2 1 -> 0, then a2's and-action-procedure called, a2's new-value = 0, lambda  
procedure inserted into segment  
propagate, after-delay  
a2's (lambda() (set-signal! output new-value)) called, set output = 0  
a1's (lambda() (set-signal! output new-value)) called, set output = 1  
  
so after this propagation we got output = 1, which is not correct

LinYihai

If someone wants to play with LIFO, alter propagate as follows:

```
(define (propagate)
  (if (empty-agenda? the-agenda)
    'done
    (let ((first-item (first-agenda-item the-agenda)))
      (remove-first-agenda-item! the-agenda)
      (propagate)
      (first-item))))
```



# sicp-ex-3.33

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

<< Previous exercise (3.32) | Index | Next exercise (3.34) >>

---

```
(define (averager a b c)
  (let ((u (make-connector))
        (v (make-connector)))
    (adder a b u)
    (multiplier c v u)
    (constant 2 v)
    'ok))
```

bro\_chenzox

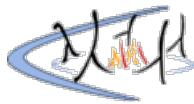
```
(define (averager a b c)
  (let ((u (make-connector)) (v (make-connector)))
    (adder a b u)
    (multiplier v c u)
    (constant 2 v)
    'ok))

(define A (make-connector))
(define B (make-connector))
(define C (make-connector))
(averager A B C)

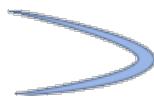
(set-value! A 100 'user)
(set-value! B 0 'user)
; 50
```

---

Last modified : 2022-12-10 17:37:49  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-3.34



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.33) | Index | Next exercise (3.35) >>

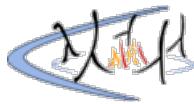
meteorgan

**if** given the following code:  
(set-value! b 4 'user)  
you will find that you can't get the value of a. Because in procedure multiplier, you only have (has-value? b) true, m1, m2 are **not** set, even they are the same, but multiplier doesn't know that.

master

Furthermore, let's imagine that there were some way for the underlying constraint to only have either m1 or m2 set, then squarer would calculate the wrong result (it would set either m1 or m2 to  $b/m^*$  instead of just the other value. Also, squarer can calculate the value of a given only b (sqrt) which multiplier cannot do.

Last modified : 2021-01-10 17:52:17  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-3.35

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

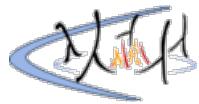
Search:

<< Previous exercise (3.34) | Index | Next exercise (3.36) >>

```
;; Squarer Constraint

(define (squarer a b)
  (define (process-new-value)
    (if (has-value? b)
        (if (< (get-value b) 0)
            (error "square less than 0 -- SQUARER" (get-value b))
            (set-value! a
                        (sqrt (get-value b)))
            me))
        (if (has-value? a)
            (set-value! b
                        (square (get-value a)))
            me)))
  (define (process-forget-value)
    (forget-value! a me)
    (forget-value! b me)
    (process-new-value))
  (define (me request)
    (cond ((eq? request 'I-have-a-value)
           (process-new-value))
          ((eq? request 'I-lost-my-value)
           (process-forget-value))
          (else
           (error "Unknown request -- SQUARER" request))))
  (connect a me)
  (connect b me)
  me)
```

Last modified : 2019-02-13 05:57:04  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-3.36

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

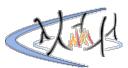
Search:

---

[<< Previous exercise \(3.35\)](#) | [Index](#) | [Next exercise \(3.37\) >>](#)

---

Last modified : 2019-03-18 00:55:01  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



## sicp-ex-3.37

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.36) | Index | Next exercise (3.38) >>

```
; ; ex-3.37

(define (c+ x y)
  (let ((z (make-connector)))
    (adder x y z)
    z))

(define (c- x y)
  (let ((z (make-connector)))
    (adder z y x)
    z))

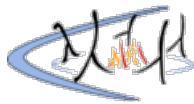
(define (c* x y)
  (let ((z (make-connector)))
    (multiplier x y z)
    z))

(define (c/ x y)
  (let ((z (make-connector)))
    (multiplier z y x)
    z))

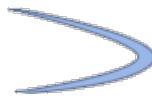
(define (cv x)
  (let ((z (make-connector)))
    (constant x z)
    z))

(define (celsius-fahrenheit-converter x)
  (+ (* (/ x 9) 5)
     32))
```

Last modified : 2013-11-07 02:46:22  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-3.38



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.37) | Index | Next exercise (3.39) >>

a)

If no interleaving is possible the resulting values can be:

45: Peter +10; Paul -20; Mary /2

35: Peter +10; Mary /2; Paul -20

45: Paul -20; Peter +10; Mary /2

50: Paul -20; Mary /2; Peter +10

40: Mary /2; Peter +10; Paul -20

40: Mary /2; Paul -20; Peter +10

b) There are  $9!/(3!3!3!) = 1680$  possible timing diagrams.

The following procedure yields all possible values.

```
(define cnt 1)
(define (execute-list lst)
  (display cnt)
  (display ":")
  (set! cnt (inc cnt))
  (define (iter lst)
    (if (null? lst)
        (newline)
        (begin ((car lst)) (iter (cdr lst)))))
  (iter lst))

(define (factorial n)
  (if (= n 0) 1 (* n (factorial (dec n)))))

(define balance 100)
(define (make-person)
  (define mybalance 100)
  (define (access)
    (set! mybalance balance))
  (define (deposit x)
    (set! mybalance (+ mybalance x)))
  (define (withdraw x)
    (set! mybalance (- mybalance x)))
  (define (withdraw-half)
    (set! mybalance (/ mybalance 2)))
  (define (sync)
    (set! balance mybalance))
  (define (check)
    (display mybalance)
    (newline)))
  (define (dispatch m)
    (cond ((eq? m 'access) access)
          ((eq? m 'deposit) deposit)
          ((eq? m 'withdraw) withdraw)
          ((eq? m 'withdraw-half) withdraw-half)
          ((eq? m 'sync) sync)
          ((eq? m 'check) check)))
  dispatch)

(define petter (make-person))
(define paul (make-person))
(define mary (make-person))

(define petter-seq (list (petter 'access) (lambda () ((petter 'deposit) 10)) (lambda () ((petter
'sync)))))
(define paul-seq (list (paul 'access) (lambda () ((paul 'withdraw) 20)) (lambda () ((paul
'sync)))))
(define mary-seq (list (mary 'access) (lambda () ((mary 'withdraw-half)))) (lambda () ((mary
'sync))))
```

```

(define result '())
(define (interleave petter paul mary temp)
  (if (and (null? petter)
            (null? paul)
            (null? mary))
      (set! result (cons (reverse temp) result)))
  (if (not (null? petter))
      (interleave (cdr petter) paul mary (cons (car petter) temp)))
  (if (not (null? paul))
      (interleave petter (cdr paul) mary (cons (car paul) temp)))
  (if (not (null? mary))
      (interleave petter paul (cdr mary) (cons (car mary) temp)))))

(interleave petter-seq paul-seq mary-seq '())
(define (in) (display balance) (set! balance 100))
(define op (map (lambda (x) (append x (list in))) result))
(for-each execute-list op)

```

Sphinxsky

I think it's 90 possibilities.

```

(define (count-demo . arrays)
  (let ((demo-result '()))

    (define (rec result arrays)
      (let ((arys (filter
                   (lambda (ary) (not (null? ary)))
                   arrays)))
        (if (null? arys)
            (set! demo-result (cons result demo-result))
            (for-each
              (lambda (ary)
                (rec
                  (append result (list (car ary)))
                  (map
                    (lambda (other)
                      (if (eq? other ary)
                          (cdr other)
                          other)))
                  arys)))
              arys))))
      (rec '() arrays)
      demo-result))

    (define (set-balance! new)
      (put 'bank 'balance new))
    (define (get-balance)
      (get 'bank 'balance))

    (define (make-process-2 name f)
      (put name 'read
           (lambda ()
             (put name 'now (get-balance)))))

    (put name 'write
         (lambda ()
           (set-balance! (f (get name 'now)))))

    (list (cons name 'read) (cons name 'write)))

    (define (make-process-3 name f)
      (put name 'read-1
           (lambda ()
             (put name 'now-1 (get-balance)))))

    (put name 'read-2
         (lambda ()
           (put name 'now-2 (get-balance)))))

    (put name 'write
         (lambda ()
           (set-balance! (f (get name 'now-1) (get name 'now-2))))))

    (list (cons name 'read-1) (cons name 'read-2) (cons name 'write)))

    (define (make-possibility demo)
      (define (get-proc k-v)
        (get (car k-v) (cdr k-v)))

      (define (make-info k-v)
        (string-append
          (symbol->string (car k-v)))

```

```

" "
  (symbol->string (cdr k-v)))))

(define (add-info info1 info2)
  (string-append
    info1
    ">"
    info2))

(let ((proc-list (map get-proc demo))
      (result 0))
  (set-balance! 100)
  ;(set-balance! 10)
  (for-each (lambda (proc) (proc)) proc-list)
  (set! result (get-balance))
  (lambda (m)
    (cond ((eq? m 'result) result)
          ((eq? m 'info)
           (accumulate add-info "\b\b\t" (map make-info demo)))
          (else (error "Unknown operation -- MAKE-RESULT" m)))))

(define (make-statistician demos)
  (let ((possibility-list (map make-possibility demos))
        (result-list '()))
    (define (iter possibility-list)
      (if (null? possibility-list)
          'done
          (let ((possibility (car possibility-list)))
            (let ((result (possibility 'result))
                  (info (possibility 'info)))
              (if (memq result result-list)
                  (put 'result result (cons info (get 'result result)))
                  (begin
                    (set! result-list (cons result result-list))
                    (put 'result result (list info))))
              (iter (cdr possibility-list)))))

    (iter possibility-list)
    (lambda (m)
      (cond ((eq? m 'all) result-list)
            ((eq? m 'one)
             (lambda (result)
               (get 'result result)))
            (else (error "Unknown operation -- MAKE-STATISTICIAN" m))))))

(define (one-poss statistician result)
  (for-each
    (lambda (info)
      (newline)
      (display info))
    ((statistician 'one) result)))

(define (all-poss statistician)
  (statistician 'all))

(define Peter (make-process-2 'Peter (lambda (x) (+ x 10))))
(define Paul (make-process-2 'Paul (lambda (x) (- x 20))))
(define Mary (make-process-2 'Mary (lambda (x) (/ x 2)))))

(define S (make-statistician (count-demo Peter Paul Mary)))

;; read all possibilities
(all-poss S)
;; (35 55 45 110 50 80 90 30 60 40)

;; read one possibility of 45
(one-poss S 45)
;; peter_read->peter_write->paul_read->paul_write->mary_read->mary_write
;; paul_read->paul_write->peter_read->peter_write->mary_read->mary_write

```

seek

It worths noting that 65 can appear as a result. Mary's operation is (balance - (balance / 2)), NOT (balance / 2).

```

; 1. Peter changes balance to 110.
; 2. Mary reads balance for the argument of subtraction, getting 110.
; 3. Paul changes balance to 90.
; 4. Mary reads balance for the argument of division, getting 90.
; 5. Mary sets balance to (110 - (90 / 2)) = 65.

```

dzy

The process described above may have a little problem.. Mary read balance as 110, so (balance / 2) = 55 and read balance again as 90, finally ans = 90 - 55 = 45

drugkeeper

A solution without using complicated code.

```
;the list is:  
;40, 35, 45, 50. (enumerate all possibilities.)  
  
;some other values may occur if processes are interleaved are:  
;referencing the timing diagram in the text:  
;balance can be first set to 110, 80, 50, (overrides the set-balance!)  
;then a third process can still occur.  
  
;from here onwards, 110-20=90, or 110/2=55.  
;(1 set-balance! overridden, then we update balance accordingly to the third process)  
;80+10=90. 80/2=40.  
;50+10=60, 50-20=30.  
  
;if 2 set-balance! are overridden then 110, 80, 50 is reached.  
  
;thus, the other values are: 110, 80, 90, 55, 60, 30.
```

codybartfast

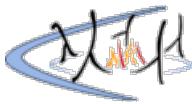
The balance can also be 25 or 70:

To get 25:

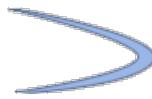
```
=====  
Peter      Paul      Mary  
          Read 100  
Read 100  
Write 110  
          Read 110  
          Write 45  
          Read 45  
          Write 25
```

To get 70:

```
=====  
Peter      Paul      Mary  
          Read 100  
Read 100  
Write 80  
          Read 80  
          Write 60  
Read 60  
Write 70
```



# sicp-ex-3.39



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.38) | Index | Next exercise (3.40) >>

adams

121 - P2 then P1  
101 - P1 then P2

100 - P1 gets  $x^2$  to be 100. Then P2 sets  $x$  after which P1 sets  $x$  to be 100.

110 cannot be done since  $x^2$  will not be interleaved with P2 which means the value of  $x$  will be the same meaning it is either 100 or 121.

11 cannot be done since once P2 starts, P1 cannot finish until P2 is done.

leafac

I believe the above is wrong and

110: P2 changes  $x$  from 10 to 11 between the two times that P1 accesses the value of  $x$  during the evaluation of  $(^ x x)$ .

is indeed possible. The serializer doesn't protect P1 at all.

atupal

@leafac I think adams is right, as "`(s (lambda () (^ x x)))`" protected the  $(^ x x)$  from interleaved, P2 can not changes  $x$  between the two times that P1 accessed the  $x$ .

karthikk

None of the above is correct!! Actually the right solution is 121, 100, 101 AND 11. adams has the right reasons for the first three. However you can also get 11: Lets label the 3 processes involved: S1: calculation of  $x^2$ , S2: setting  $x$  to calculated value of  $x^2$ ; R1: acquisition of value of  $x$  and setting it to  $x+1$ . Now here is what can happen: S1: acquires the mutex and calculates the value of  $x^2$  i.e. 100, now it releases the mutex. At this point R1 acquires the mutex and calculates a value of 11 BUT before it can set the value to 11, S2 occurs (nothing barring it from occurring concurrently with R1) and therefore R1 gets to set the final value of  $x$ , i.e. 11. You can convince yourselves by running the following simulation:

```
(define (run-many-times in-test num)
  (define (run-test numtimes output)
    (let ((m (in-test)))
      (cond ((= 0 numtimes) output)
            ((memq m output) (run-test (- numtimes 1) output))
            (true (run-test (- numtimes 1) (cons m output))))))
  (run-test num '()))

(define (test-2)
  (define x 10)
  (define s (make-serializer))
  (parallel-execute (lambda () (set! x ((s (lambda () (* x x))))))
                    (s (lambda () (set! x (+ x 1))))))
  x)

(display (run-many-times test-2 10000))

;Output will be (11 100 101 121) in some order
```

Hoonji

I agree with karthikk. Here's the event sequence that produces 11:

- P1) access  $x: 10$
- P1) new value:  $10 * 10 = 100$
- P2) access  $x: 10$
- P2) new value:  $10 + 1 = 11$
- P1) **set!  $x$  to 100**
- P2) **set!  $x$  to 11**

The key idea here is that "P1) set!  $x$  to 100" can be interleaved between P2 events.

jphilliq

@adams is right, you can't get 11. People say you can get 11 because you access x in P2, compute 11, and then P1 sets x to 100 before P2 can. THIS IS WRONG. P2, after computing  $x + 1$ , CANNOT BE INTERRUPTED BY P1. This is the entire point of serialization, or there would be no point. So it sets x to 11, without P1 setting x, and then x is set to 100. 11 IS NOT A POSSIBLE VALUE.

eric4brs

@jphilliq says that P2 cannot be interrupted by P1. This is wrong. Serialization is cooperative. Just because all of P2 is serialized does not protect it from being interleaved with a non-serialized process. At first blush it appears that there are 3 things to consider as being interleavable:

- P1a Access and compute ( $* x x$ )
- P1b Set x to computed ( $* x x$ )
- P2 Access and compute and set x to ( $+ x 1$ )

However, since P1b is not serialized, it can occur at any point along P2. The serialization of P2 has no effect on P1b. P2 is made up of 2 atomic operations:

- P2a read and calculate ( $+ x 1$ )
- P2b set x to computed ( $+ x 1$ )

P1a cannot interleave between P2a and P2b, but P1b can. That yields the following possibilities:

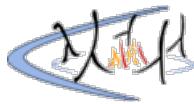
- P1a P1b P2a P2b --> (\* 10 10) 100->x (+ 100 1) 101->x : 101
- P2a P2b P1a P1b --> (+ 10 1) 11->x (\* 101 101) 121->x : 121
- P1a P2a P1b P2b --> (\* 10 10) (+ 10 1) 100->x 11->x : 11
- P1a P2a P2b P1b --> (\* 10 10) (+ 10 1) 11->x 100->x : 100

@karthikk and @Hoonji are correct.

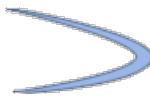
krubar

Assuming that what serializer does in essence is making sure that two (or more) functions having the same serializer will run sequentially, then the execution of P1 (not serialized) and P2 (serialized) can still be interleaved and possible results are the same as if P2 was not serialized at all.

Then possible values are the same as for running in parallel P1 and P2 without any serialization: 101, 121, 110, 11, 100.



# sicp-ex-3.40



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.39) | Index | Next exercise (3.41) >>

meteorgan

```
1,000,000: p1 set x to 100, then p2 set x to x*x*x. or p2 set x to 1,000, then p1 set x to x*x.  
10,000: p1 get x = 10, then p2 set x to 1000, then p1 compute 10*1000. or p2 get x, compute x*x = 100, then p1 set x to 100, then p2 set x to 100*100.  
100: p1 compute x*x = 100, then p2 set x to 10*10*10, then p1 set x to 100.  
100,000: p2 get x = 10, then p1 set x to 100, then p2 set x to 10*100*100.  
1000: p2 compute 10*10*10 = 1000, then p1 set x to 100, then p2 set x to 1000.  
  
after serializing, only 1,000,000 exists.
```

Hoonji

Elaboration on the same answer:

```
P1: (lambda () (set! x (* x x)))  
P2: (lambda() (set! x (* x x)))
```

\*\* "x1", "x2", "x3" below refer to the first, second, and third arguments of the respective procedures

P1 events:  
a) access x1  
b) access x2  
c) new value = x1 \* x2  
d) set! x to new value

P2 events:  
v) access x1  
w) access x2  
x) access x3  
y) new value = x1 \* x2 \* x3  
z) set! x to new value

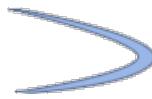
significant event sequences and their results:

\*\* (P1) refers to event sequence "abcd", (P2) refers to event sequence "vwxyz"  
1. (P1)(P2) => 1,000,000  
2. (P2)(P1) => 1,000,000  
3. a(P2)bc => 10,000  
4. v(P1)wxyz => 100,000  
5. vw(P1)xyz => 10,000  
6. abc(P2)d => 100  
7. vwxy(P1)z => 1,000

Unique results:  
[100, 1000, 10000, 100000, 1000000]

After serializing P1 and P2:  
Only event sequence 1 and 2 remain:  
[1000000]

# sicp-ex-3.41



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.40) | Index | Next exercise (3.42) >>

xdaividliu

serializing balance is not necessary because getting the balance \*doesn't change the state of the account\*. Even if you serialize the balance accessor, a parallel execution of a series of withdraw!'s deposit!'s and balance's will still occur in an undefined order. Hence, any ambiguities that would occur when the balance accessor is \*not\* serialized would still occur if it \*is\* serialized.

When balance is not serialized, then calls to balance may interleave into a call to withdraw! or deposit!. However, since there is only one call to set! in each of these functions, balance will still either get only the "before" or "after" values of the balance, and not some "intermediate" value that is neither the value before or after the withdraw! or deposit!

Hence, whether we serialize the balance accessor or not, we will always get the same results.

meteorgan

No, **if** you get balance in different order with withdraw **and** deposit, you will get different result. but all they all legal.

mas

I disagree. I think the answer is yes. Another process could access balance mid-withdraw or mid-deposit and get the wrong info.

Shawn

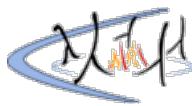
It's not wrong info. If a withdraw or deposit isn't finished, then the original balance is still valid. Accessing balance is a single operation, it cannot be interleaved with other procedures. Therefore, we don't need to protect it.

Mike

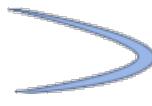
I believe the book uses these two requirements on concurrency:

- First, it does not require the processes to actually run sequentially, but only to produce results that are the same as if they had run sequentially.
- Second, there may be more than one possible ``correct'' result produced by a concurrent program, because we require only that the result be the same as for some sequential order.

An "unprotected" access to balance fulfills both requirements so I would say it does not really matter.



# sicp-ex-3.42



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.41) | Index | Next exercise (3.43) >>

xdavidliu

A possible test for whether the change proposed here is safe:

```
;  
(let ((acc (make-account 100)))  
  (parallel-execute  
    (lambda () ((acc 'withdraw) 10))  
    (lambda () ((acc 'withdraw) 20)))  
  (acc 'balance))
```

The original version in book \*certainly\* returns 70 here, since each of the two withdrawals results in a \*separately\* serialized procedure. The modified version on the other hand has both withdrawals using a \*single\* serialized procedure.

Without further knowledge of how make-serial and parallel-execute actually work, it's hard to tell whether a serialized function can be interleaved \*with itself\*. If yes, then for the modified version, there may be a data race and the above block of code can incorrectly return 90 or 80. If not, then the modified version in this exercise works perfectly.

A reasonable thing to say would be that since both arguments to parallel-execute were serialized using the same serializer (since they are actually the same \*function\*), we should expect them to not interleave with each other, and hence the modification \*should\* work.

beckett

it is a safe change to make, for it will get the same final result as the original version. we should differentiate the code copy, which is the code to be run, to runtime copy, which is the process running to be executed. the original version will execute serializer function each time the account receiving a message, which will generate a \*separate runtime copy\* as a new sub-process from the \*withdraw/deposit code\*. While the modified version will generate a \*mediate code copy\* at the time as the account constructed in the main process, from which it will generate a \*separate runtime copy\* as a new sub-process each time the account receiving a message.

meteorgan

It's safe to do that change. There is nothing different about concurrency in these two version. The only difference is new one serialize the procedures before call the functions, but the original one do it when call withdraw or deposit.

The original solution and the one proposed by Ben Bitdiddle are essentially the same.

squarebat

That is not the only difference. Every call to serializer returns a new function that accesses the same serializer. With Ben Bitdiddle's version, only two serialized functions are created - one for withdraw and deposit. With the original solution, a new serialized function is created on each call to withdraw or deposit. As described by xdavidliu, the original is the safer solution, since it's unclear whether a function would interleave with itself. It's entirely possible, since each function call creates a new process which are treated differently.

jeffg

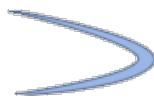
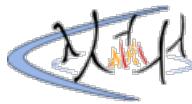
I agree with it's safe to change, but I don't think there is no difference. In changed version, all accounts created share a same serialized set, which means only one account could execute withdraw or deposit at a same time. Hence there will be no concurrency between different accounts.

eric4brs

Every time make-account is called make-serializer is called. There will be a unique serializer for every account. I don't agree that "only one account could execute withdraw or deposit at the same time."



# sicp-ex-3.43



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.42) | Index | Next exercise (3.44) >>

asmn

Disagree that sum of balances must remain the same in the first exchange program.

If we have three accounts A B, C with balances 10,20,30 respectively:

by running P1: (exchange C A) and P2:(exchange C A) concurrently this may happen:

1. the difference is set to 20 on both P1 and P2.
2. 20 is withdrawn from C and 20 is deposited to A in P1 Now the balances are C->10 A->30 B->20
3. 20 is withdrawn from C again, giving the Insufficient funds message with no money taken. 20 is deposited to A again Now the balances are C->10 A->50 B->20

the total has increased from 60 to 80. Forgive me if I have missed something.

timothy235

Double serializing the exchange procedure, using both accounts' serializers, protects the exchange operations from being interleaved with other account operations from either account. This makes exchange atomic. So its behavior will be as expected.

With the original definition of exchange, the exchange operation will not be atomic, but it will be composed of exactly four atomic operations, namely read the two account balances, make an atomic withdrawal from the larger account, and make an atomic deposit into the smaller account.

To see how the original exchange definition might not preserve the three balances, consider the following three exchanges:

- acct1 = 10
- acct2 = 20
- acct3 = 30
- P1: exchange accounts 1 and 2
- P2: exchange accounts 2 and 3
- P3: exchange accounts 1 and 3

The original exchange allows us to interleave P3 between reads of P1 and P2:

- P1 reads acct1 = 10
- P2 reads acct2 = 20
- P3 makes acct1 = 30 and acct3 = 10
- P1 reads acct2 = 20
- P2 reads acct3 = 10
- P1 thinks acct1 = 10 (but it's really 30) and acct2 = 20  
So P1 adds 10 to acct1 and subtracts 10 from acct2  
Now acct1 = 40 and acct2 = 10
- P2 thinks acct2 = 20 (but it's really 10) and acct3 = 10  
So P2 subtracts 10 from acct2 and adds 10 to acct3  
Now acct2 = 0 and acct3 = 20

Final balances:

- acct1 = 40
- acct2 = 0
- acct3 = 20

However the sum of all three balances cannot change. This is because each exchange adds and removes an equal amount between accounts. So total deposits will always equal total withdrawals.

Sphinxsky

You can perform the following simulation to verify:

```

(define (count-demo . arrays)
  (let ((demo-result '()))

    (define (rec result arrays)
      (let ((arys (filter
                    (lambda (ary) (not (null? ary)))
                    arrays)))
        (if (null? arys)
            (set! demo-result (cons result demo-result))
            (for-each
              (lambda (ary)
                (rec
                  (append result (list (car ary))))
                  (map
                    (lambda (other)
                      (if (eq? other ary)
                          (cdr other)
                          other)))
                  arys)))
            (rec '() arrays)
            demo-result)))

(define (make-account name balance)
  (put 'balance name balance)
  (lambda (m)
    (cond ((eq? m 'set-balance!)
           (lambda (new) (put 'balance name new)))
          ((eq? m 'get-balance)
           (get 'balance name))
          (else (error "Unknown operation -- MAKE-ACCOUNT" m)))))

(define (set-balance! acc new)
  ((acc 'set-balance!) new))
(define (get-balance acc)
  (acc 'get-balance))

(define a1 (make-account 'a1 10))
(define a2 (make-account 'a2 20))
(define a3 (make-account 'a3 30))

(define (reset-accounts)
  (set-balance! a1 10)
  (set-balance! a2 20)
  (set-balance! a3 30))

(define (get-balances-all)
  (let ((a1 (get-balance a1))
        (a2 (get-balance a2))
        (a3 (get-balance a3)))
    (string->symbol
      (string-append
        "("
        (string a1)
        ")" + (""
        (string a2)
        ")" + (""
        (string a3)
        ")" = (""
        (string (+ a1 a2 a3))
        ")" )))))

(define (make-exchange name acc1 acc2)
  (put name 'read_acc1
    (lambda ()
      (put name 'acc1 (get-balance acc1)))))

  (put name 'read_acc2
    (lambda ()
      (put name 'acc2 (get-balance acc2)))))

  (put name 'write_acc1
    (lambda ()
      (set-balance!
        acc1
        (- (get-balance acc1)
            (- (get name 'acc1)
                (get name 'acc2))))))

  (put name 'write_acc2
    (lambda ()
      (set-balance!
        (- (get-balance acc2)
            (- (get name 'acc2)
                (get name 'acc1)))))))

```

```

acc2
(+ (get-balance acc2)
   (- (get name 'acc1)
       (get name 'acc2)))))

(list (cons name 'read_acc1) (cons name 'read_acc2) (cons name 'write_acc1) (cons
name 'write_acc2)))
```

```

(define (make-possibility demo)
  (define (get-proc k-v)
    (get (car k-v) (cdr k-v)))

  (define (make-info k-v)
    (string-append
      (symbol->string (car k-v))
      " "
      (symbol->string (cdr k-v)))))

  (define (add-info infol info2)
    (string-append
      infol
      ">>"
      info2))

  (let ((proc-list (map get-proc demo)))
    (result '())
    (reset-accounts)
    (for-each (lambda (proc) (proc)) proc-list)
    (set! result (get-balances-all))
    (lambda (m)
      (cond ((eq? m 'result) result)
            ((eq? m 'info)
              (accumulate add-info "\b\b\t" (map make-info demo)))
            (else (error "Unknown operation -- MAKE-RESULT" m))))))

(define (make-statistician demos)
  (let ((possibility-list (map make-possibility demos))
        (result-list '()))
    (define (iter possibility-list)
      (if (null? possibility-list)
          'done
          (let ((possibility (car possibility-list)))
            (let ((result (possibility 'result))
                  (info (possibility 'info)))
              (if (memq result result-list)
                  (put 'result result (cons info (get 'result result)))
                  (begin
                    (set! result-list (cons result result-list))
                    (put 'result result (list info))))
              (iter (cdr possibility-list)))))

      (iter possibility-list)
      (lambda (m)
        (cond ((eq? m 'all) result-list)
              ((eq? m 'one)
               (lambda (result)
                 (get 'result result)))
              (else (error "Unknown operation -- MAKE-STATISTICIAN" m))))))

  (define (one-poss statistician result)
    (let ((count 0))
      (for-each
        (lambda (info)
          (newline)
          (display info)
          (set! count (+ 1 count)))
        ((statistician 'one) result))
      count))

  (define (all-poss statistician)
    (statistician 'all))

  (define Peter (make-exchange 'Peter a1 a2))
  (define Paul (make-exchange 'Paul a1 a3))

  (define S (make-statistician (count-demo Peter Paul)))

; Run the following statement to prove the middle two questions
(all-poss S)
; Possible results
; "(30)+(10)+(20)=(60)" "(40)+(10)+(10)=(60)" "(20)+(30)+(10)=(60)"
; The ratio of their occurrence times 5:6:5
```

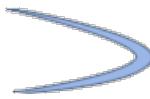
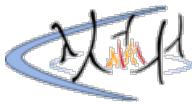
```

; =====
; If you want to answer the last question
; The following changes are required
; The rest of the code remains the same
(define (make-exchange name acc1 acc2)
  (put name 'read_acc1
    (lambda ()
      (put name 'acc1 (get-balance acc1))))
  (put name 'read_acc2
    (lambda ()
      (put name 'acc2 (get-balance acc2))))
  (put name 'write_r_acc1
    (lambda ()
      (put name 'write_r_d_acc1 (get-balance acc1))))
  (put name 'write_w_acc1
    (lambda ()
      (set-balance!
        acc1
        (- (get name 'write_r_d_acc1)
            (- (get name 'acc1)
                (get name 'acc2))))))
  (put name 'write_r_acc2
    (lambda ()
      (put name 'write_r_d_acc2 (get-balance acc2))))
  (put name 'write_w_acc2
    (lambda ()
      (set-balance!
        acc2
        (+ (get name 'write_r_d_acc2)
            (- (get name 'acc1)
                (get name 'acc2))))))

(list
  (cons name 'read_acc1)
  (cons name 'read_acc2)
  (cons name 'write_r_acc1)
  (cons name 'write_w_acc1)
  (cons name 'write_r_acc2)
  (cons name 'write_w_acc2))

; Possible results
; "(30)+(10)+(20)=(60)" "(30)+(10)+(10)=(50)" "(20)+(10)+(10)=(40)" "(40)+(10)+(10)=(60)"
; "(20)+(30)+(10)=(60)"
; The ratio of their occurrence times 28:200:200:468:28

```



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.43) | Index | Next exercise (3.45) >>

erben

Exchange requires both accounts to be idle before we start exchanging (i.e. exchanging a with b while a is still exchanging with c is not allowed)

For transfer, there is no such requirement. Louis is wrong.

sam

In the exchange problem, state of one account depends on the state of another account (they are coupled). There is no such coupling in the transfer problem.

karthikk

Another way to rephrase what sam says is to say that exchange has an unserialized access (the difference process) which can occur concurrently with another exchange underway while transfer has no such process...

xdaavidliu

@gsanghera: Louis P. Reasoner is a strawman character created by the authors to always provide loose reasoning, and hence is always supposed to be wrong.

gsanghera

Louis is right. There is an issue with Ben's logic. If you run multiple transfers simultaneously between multiple accounts, in Ben's version it is possible that accounts run out of money without needing to - failing some transactions. In Louis' version, as the full exchange is serialized, money must enter the account to which it's being transferred, before another transfer can be initiated from it. If you have a lot of money, it's not a problem (you will eventually be credited). If you have a little money, you might not be so indifferent.

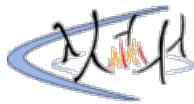
e.g. A has 10, B 20, C 10  
T1 -> 10 from C to B,  
T2 -> 30 from B to A

If T1 and T2 are fully serialized, T2 will proceed. Otherwise while T1 is half completed (10 leaves C) T2 could initiate, and fail. After this B gets the additional 10, but it's too late.

Note that if T1 and T2 are initiated simultaneously, using parallel-execute, T2 could still fail if it initiates before T1. But in general, it will still be much better to ensure money that leaves an account enters another account before another transaction is started on the 2nd account. In other words - a transfer should be atomic.

dzy

this result is same as a sequential process, in which B -> A first then C -> B , so it's correct.



[\*\*<< Previous exercise \(3.44\) | Index | Next exercise \(3.46\) >>\*\*](#)

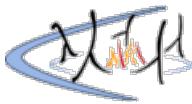
xdavidliu

The correct implementation of serialized-exchange in the book deliberately leaves the dispatched withdraw and deposit procedures \*un\*-serialized, so that when the "raw" exchange procedure calls them and then gets wrapped, via serialized-exchange, in the serializers of both accounts involved in the exchange, there will be no conflict.

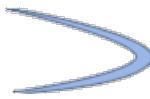
Louis' proposed change would be disastrous because the dispatched withdraw and deposit procedures called in the "raw" exchange procedure would \*already\* have serializers, so when serialized-exchange wraps the raw exchange in both serializers \*again\*, it wouldn't even be possible to perform the required withdraws and deposits, \*by definition\*, since two procedures can be run concurrently if and only if they have \*not\* been serialized with the same serializer.

erben

Using Louis's code, in exchanging two accounts we will use the same serializer for two times. One in the serialized-exchange function and the other in the dispatch function. According to the implementation of the serializer, if a process has already acquired a mutex and it wants to acquire that mutex again, the busy waiting will never halt.



# sicp-ex-3.47



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.46) | Index | Next exercise (3.48) >>

gws

```
(define (make-semaphore n)
  (let ((lock (make-mutex))
        (taken 0))
    (define (semaphore command)
      (cond ((eq? command 'acquire)
             (lock 'acquire)
             (if (< taken n)
                 (begin (set! taken (+ taken)) (lock 'release))
                 (begin (lock 'release) (semaphore 'acquire))))
            ((eq? command 'release)
             (lock 'acquire)
             (set! taken (- taken))
             (lock 'release)))
            (else
             (error "Unknown semaphore command")))))
    (semaphore)))
```

leafac

I think the above implementation can be improved. Instead of the explicit busy way, we can use a second mutex that makes the clients hang. This way, if we can come up with a better implementation for mutexes, semaphores get the benefit as well:

```
(define (make-semaphore maximum-clients)
  (let ((access-mutex (make-mutex))
        (exceeded-mutex (make-mutex))
        (clients 0))
    (define (the-semaphore message)
      (cond ((eq? message 'acquire)
             (access-mutex 'acquire)
             (cond ((> clients maximum-clients)
                    (access-mutex 'release)
                    (exceeded-mutex 'acquire)
                    (the-semaphore 'acquire))
                (else
                  (set! clients (+ clients 1))
                  (if (= clients maximum-clients)
                      (exceeded-mutex 'acquire)
                      (access-mutex 'release)))))
            ((eq? message 'release)
             (access-mutex 'acquire)
             (set! clients (- clients 1))
             (exceeded-mutex 'release)
             (access-mutex 'release)))
            (else
              (error "Unknown semaphore message")))))
    (the-semaphore)))
```

dzy

I think this solution can get stuck, suppose we run three process A, B, C. A and B are "acquire", C is "release". A is waiting for exceeded-mutex after release its access-mutex. and C get access-mutex and release exceeded-mutex and A get exceeded-mutex, and C release access-mutex and B get it. Then B add the client and require exceeded-mutex but A get it. So now A seeks for access-mutex and B seeks for exceeded-mutex and will infinitely wait.

Also, here's an answer for part b, which is very similar to what gws did. Here, the busy wait is inevitable:

```
(define (make-semaphore maximum-clients)
  (let ((access-mutex (list false))
        (clients 0))
    (define (the-semaphore message)
      (cond ((eq? message 'acquire)
             (if (test-and-set! access-mutex)
                 (the-semaphore 'acquire))
            ((cond ((> clients maximum-clients)
                   (clear! access-mutex)
                   (the-semaphore 'release)))))))
```

```

        (the-semaphore 'acquire))
(else
  (set! clients (+ clients 1))
  (clear! access-mutex)))
((eq? message 'release)
 (if (test-and-set! access-mutex)
     (the-semaphore 'release))
  (set! clients (- clients 1))
  (clear! access-mutex)))
the-semaphore))

```

djrochford

This is a slightly simpler version of the acquire part of leafac's mutex solution which I \*think\* is still correct (delete me if I'm wrong).

```

(exceed-mutex 'acquire)
(access-mutex 'acquire)
(set! clients (+ clients 1))
(if (< clients maximum-clients)
    (exceed-mutex 'release))
(access-mutex 'release)

```

Sphinxsky

```

(define (make-cell-cycle n)

  (define (rec-make n)
    (if (= n 0)
        '()
        (cons #f (rec-make (- n 1)))))

  (let ((cells (rec-make n)))
    (set-cdr! (last-pair cells) cells)
    cells))

(define (retry-acquire cell)
  (if (test-and-set! cell)
      (retry-acquire (cdr cell)))))

(define (retry-release cell)
  (if (car cell)
      (clear! cell)
      (retry-release (cdr cell))))
; b)
; The acquire operation must precede the release operation.
(define (make-semaphore-use-tas n)
  (let ((cells (make-cell-cycle n)))
    (define (the-semaphore m)
      (cond ((eq? m 'acquire) (retry-acquire cells))
            ((eq? m 'release) (retry-release cells))
            (else (error "Unknown operation -- MAKE-SEMAPHORE" m))))
    the-semaphore))

; a)
(define (make-semaphore-use-mutex n)
  (let ((lock (make-mutex))
        (clients n))
    (define (the-semaphore m)
      (cond ((eq? m 'acquire)
             (lock 'acquire)
             (if (> clients 0)
                 (begin
                   (set! clients (- clients 1))
                   (lock 'release))
                 (begin
                   (lock 'release)
                   (the-semaphore 'acquire))))
             ((eq? m 'release)
              (lock 'acquire)
              (set! clients (+ clients 1))
              (lock 'release))
             (else (error "Unknown operation -- MAKE-SEMAPHORE" m)))))
    the-semaphore)))

```

fry

No busy waiting and hopefully no deadlock in part a)

Part b) included for kicks.

```

(define (make-semaphore max)
  (define processes 0)
  (define accessing (make-mutex))
  (define waiting (make-mutex))
  (define (acquire)
    (if (< processes max)
        (begin (set! processes (+ processes 1))
               (if (= processes max)
                   (waiting 'acquire))
               (accessing 'release))
        (begin (accessing 'release)
               (waiting 'acquire)
               (dispatch 'acquire))))
  (define (release)
    (if (> processes 0)
        (set! processes (- processes 1))
        (waiting 'release)
        (accessing 'release)))
  (define (dispatch m)
    (accessing 'acquire)
    (cond ((eq? m 'acquire) (acquire))
          ((eq? m 'release) (release))))
  dispatch)

(define (make-semaphore max)
  (let ((processes 0)
        (access (list false)))
    (define (dispatch m)
      (cond ((eq? m 'acquire) (acquire))
            ((eq? m 'release) (release))))
    (define (acquire)
      (cond ((test-and-set! access) (acquire))
            ((< processes max)
             (set! processes (+ processes 1))
             (clear! access))
            (else (clear! access) (acquire))))
    (define (release)
      (cond ((test-and-set! access) (release))
            ((> processes 0)
             (set! processes (- processes 1))
             (clear! access))
            (else (clear! access))))
    dispatch))

```

rohitkg98

we have the serializer we implement using a mutex, can't we just use that?

edit: my bad, this will result in a deadlock upon hitting max due to not being able to run release

```

; the acquiring and releasing of mutexes need to be serialized
; this will go into a deadlock upon hitting max mutexes
; the release and acquire cannot be on the same serializer
; bad solution

(define (make-semaphore n)
  (let ((total 0)
        (serializer (make-serializer)))
    (define (acquire)
      (if (< total n)
          (set! total (+ total 1))
          (acquire)))
    (define (release)
      (set! total (- total 1)))
    (define (the-semaphore m)
      (cond ((eq? m 'acquire) (serializer acquire)) ; retry
            ((eq? m 'release) (serializer release))))))

```

Correct Solutions with explanation:

```

; a correct solution would acquire lock and release before retrying
; so that release also gets a chance
(define (make-semaphore n)
  (let ((total 0)
        (access-lock (make-mutex)))
    (define (acquire)
      (access-lock 'acquire)
      (if (< total n)
          (begin (set! total (+ total 1))
                 (access-lock 'release))
          (begin (access-lock 'release)

```

```

        (acquire)))
(define (release)
  (access-lock 'acquire)
  (set! total (- total 1))
  (access-lock 'release))
(define (the-semaphore m)
  (cond ((eq? m 'acquire) (acquire))
        ((eq? m 'release) (release)))))

; now, an even better way would be create separate locks for access and max
; this way if a process got an access lock but max value is reached
; they release access lock and go into max lock
; as soon as max opens up when release gets called they continue execution by calling
acquire again
(define (make-semaphore n)
  (let ((total 0)
        (access-lock (make-mutex))
        (release-indicator (make-mutex)))
    (define (acquire)
      (access-lock 'acquire)
      (if (< total n)
          (begin (set! total (+ total 1))
                 (access-lock 'release))
          ; all semaphores occupied
          ; don't proceed further until one is released
          (begin (access-lock 'release)
                 (release-indicator 'acquire)
                 (acquire)))
      (define (release)
        (access-lock 'acquire)
        (set! total (- total 1))
        ; indicate that there is atleast one free semaphore now
        (release-indicator 'release)
        (access-lock 'release))
      (define (the-semaphore m)
        (cond ((eq? m 'acquire) (acquire))
              ((eq? m 'release) (release)))))))

```

x3v

Not sure if this solution falls into part a or part b. In this implementation, the cell value of the mutex will only be set to #t if max\_processes is reached. Otherwise, a call to acquire the semaphore will continue looping until the number of active processes is decremented below the max.

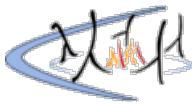
Honestly, in this implementation, the mutex procedure object isn't even strictly required - just need a local variable to store the boolean, or perhaps just use the running count of active processes?

Please let me know if this is correct, any and all comments are more than welcome.

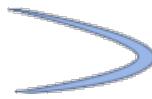
```

(define (make-semaphore max_processes)
  (let ((num_processes 0)
        (mutex (make-mutex)))
    (define (maxed-processes?)
      (= num_processes max_processes))
    (define (test-and-set!)
      (if (maxed-processes?) #t
          (begin
            (set! num_processes (+ num_processes 1))
            (if (maxed-processes?) (mutex 'acquire))
            #f)))
    (define (clear!)
      (if (maxed-processes?)
          (mutex 'release))
      (if (= num_processes 0)
          (error "Num processes already at minimum")
          (set! num_processes (- num_processes 1))))
    (define (the-semaphore m)
      (cond ((eq? m 'acquire) (if (test-and-set!) (the-semaphore 'acquire)))
            ((eq? m 'release) (clear!))
            (else (error "Unknown request" m))))
      the-semaphore)))

```



# sicp-ex-3.48



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.47) | Index | Next exercise (3.49) >>

gws

```
(define (serialized-exchange account1 account2)
  (let ((serializer1 'serializer-for-bigger-id--acc)
        (serializer2 'serializer-for-smaller-id-acc))
    (cond ((> (get-id account1) (get-id account2))
           (set! serializer1 (account1 'serializer)))
          (set! serializer2 (account2 'serializer)))
          (else (set! serializer1 (account2 'serializer))
                (set! serializer2 (account1 'serializer))))
    ((serializer1 (serializer2 exchange)) account1 account2)))
```

leafac

This works because it makes impossible for one process to have acquired the lock for a resource A and be waiting for a lock for a resource B while other process has the lock for the resource B and is waiting for a lock for the resource A.

A complete solution that also implements the necessary modifications to make-account-and-serializer and avoids using state:

```
(define (make-account-and-serializer id balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((balance-serializer (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'id) id)
            ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit) deposit)
            ((eq? m 'balance) balance)
            ((eq? m 'serializer) balance-serializer)
            (else (error "Unknown request -- MAKE-ACCOUNT"
                         m))))
    dispatch))

(define (serialized-exchange account1 account2)
  (let* ((serializer1 (account1 'serializer))
        (serializer2 (account2 'serializer))
        (exchanger (if (< (account1 'id) (account2 'id))
                      (serializer1 (serializer2 exchange))
                      (serializer2 (serializer1 exchange)))))
    (exchanger account1 account2)))
```

Sphinxsky

```
(define (make-counter initial)
  (define (count-and-read)
    (set! initial (1+ initial))
    initial)
  ((make-serializer) count-and-read))
(define get-id (make-counter 0))

(define (make-account-and-serializer balance)
```

```

(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Insufficient funds"))
(define (deposit amount)
  (set! balance (+ balance amount))
  balance)
(let ((balance-serializer (make-serializer))
      (id (get-id)))
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          ((eq? m 'balance) balance)
          ((eq? m 'serializer) balance-serializer)
          ((eq? m 'id) id)
          (else (error "Unknown operation -- MAKE-ACCOUNT" m))))
  dispatch))

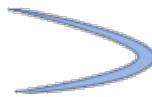
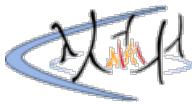
(define (deposit account amount)
  (let ((s (account 'serializer))
        (d (account 'deposit)))
    ((s d) amount)))

(define (withdraw account amount)
  (let ((s (account 'serializer))
        (w (account 'withdraw)))
    ((s w) amount)))

(define (exchange account1 account2)
  (let ((difference (- (account1 'balance) (account2 'balance))))
    ((account1 'withdraw) difference)
    ((account2 'deposit) difference)))

(define (serialized-exchange account1 account2)
  (let ((serializer1 (account1 'serializer))
        (serializer2 (account2 'serializer)))
    ((if (< (account1 'id) (account2 'id))
         (serializer2 (serializer1 exchange))
         (serializer1 (serializer2 exchange)))
     account1
     account2)))

```



<< Previous exercise (3.48) | Index | Next exercise (3.50) >>

leafac

If the process can't know all the locks it's going to need prior to requesting the first lock, there's no way for it to enforce the ordering.

xdaavidliu

The reason why the scheme from the previous exercise works to prevent deadlocks is that exchanging account balances is a symmetric operation, e.g. exchanging account A with account B is equivalent to exchanging account B with account A.

However, we can conceive of an operation that is \*not\* symmetric. For example, suppose every account has a list-of-transfer-amounts, and we wanted an operation (serialized-list-transfer account-from account-to), which transfers the amount (car (account-from 'list-of-transfer-amounts)) from account-from to account-to, and then updates the pointer of (account-from 'list-of-transfer-amounts) to the next amount.

In this case, the transfer \*must\* be serialized with account-to's serialized first (on the inside), and then serialized with account-from's serializer (on the outside); the order is \*not\* allowed to be permuted to our convenience, e.g. using id numbers or whatever. Hence, suppose one user calls (serialized-list-transfer account-a account-b) and another user calls (serialized-list-transfer account-b account-a), there is a possibility for deadlock that \*cannot\* be fixed using an id number to determine the order of serialization (since, again, the operation is not symmetric and hence the order of serialization is not freely determined).

fry

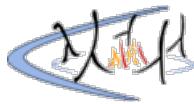
I think in this example we could still use account-from and account-to's serializers in any order.

fry

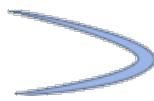
It wouldn't work for procedures that operate on a shared resource before determining some other shared resource to operate on.

Suppose accounts have an internal list of pairs consisting of an account and an amount and we invent a procedure that can take one account, and look at the first pair in its list and make a deposit accordingly.

Then if account A has account B first in its list and vice versa, concurrently calling our invented procedure on both could result in a deadlock.



# sicp-ex-3.50



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.49) | Index | Next exercise (3.51) >>

meteorgan

```
(define (stream-map proc . argstreams)
  (if (stream-null? (car argstreams))
      the-empty-stream
      (cons-stream
        (apply proc (map stream-car argstreams))
        (apply stream-map
          (cons proc (map stream-cdr argstreams))))))
```

Beckett

argstreams is a list of list, not a stream, we should not use stream-xxx procs on it.

```
(define (stream-map proc . argstreams)
  (if (stream-null? (car argstreams))
      the-empty-stream
      (cons-stream
        (proc (map car argstreams))
        (apply stream-map (cons proc (map cdr argstreams))))))
```

Patrick-OHara

That implementation assumes that the streams are the same length (perhaps infinite). In order to cope with streams of different lengths I have this:

```
(define (filter predicate list)
  (cond ((null? list)
         '())
        ((predicate (car list))
         (cons (car list) (filter predicate (cdr list))))
        (else
         (filter predicate (cdr list)))))
(define (stream-map proc . argstreams)
  (let ((non-null-args (filter (lambda (s) (not (stream-null? s))) argstreams)))
    (if (null? non-null-args)
        the-empty-stream
        (cons-stream
          (apply proc (map stream-car non-null-args))
          (apply stream-map
            (cons proc (map stream-cdr non-null-args)))))))
```

Shade

A bit cleaner:

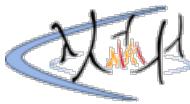
```
(define (any? pred lst)
  (cond ((null? lst) #f)
        ((pred (car lst)) #t)
        (else (any? pred (cdr lst)))))

(define (stream-map proc . streams)
  (if (any? stream-null? streams)
      empty-stream
      (stream-cons
        (apply proc (map stream-car streams))
        (apply stream-map
          proc ; We don't need to cons proc and the result of map.
          (map stream-cdr streams)))))
```

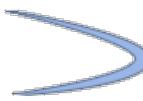
sanggggg

to Beckett, i think argstream is a list of stream not list of list. so it is correct to map it by stream-xxx. (map each stream by stream-xxx)





# sicp-ex-3.51



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.50) | Index | Next exercise (3.52) >>

perry

```
(define (show x)
  (display-line x)
  x)

(define x (stream-map show (stream-enumerate-interval 0 10)))

0
;Value: x

(stream-ref x 5)

1
2
3
4
5

;Value: 5

(stream-ref x 7)

6
7
;Value: 7
```

master

I get a different result. On Racket, with all of the procedures implemented exactly as specified in the book, I get:

```
racket@> (define x
  (stream-map show
    (stream-enumerate-interval 0 10)))

0
1
2
3
4
5
6
7
8
9
10racket@> (stream-ref x 5)
5
racket@> (stream-ref x 7)
7
```

The only thing which could possibly account for this discrepancy is a mistake in my implementation of `delay`, but it looks fine to me?

```
(define delay (lambda (proc) (memo-proc (lambda () proc))))
```

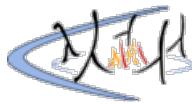
Any idea why I get different results?

EDIT: After solving the next exercise and looking at other solutions, it seems like the evaluations aren't actually being delayed, could it be some aspect of Racket's implementation which is preventing it from working?

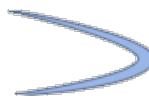
madhusudann

master, the "delay" implementation that you have provided doesn't prevent the expression from being evaluated while you call the delay function. While making the call `(delay <exp>)`: How do you convey to the interpreter that the exp should not be evaluated before calling the delay function. The book mentions that delay is equivalent to the implementation(not equal) that you have provided as it requires a special syntax rule that prevents execution while calling. I guess that is why in the book they said they are equivalent but they never said they are equal. Remove your custom implementation of delay and use the

language provided and you will see the expected behavior.



# sicp-ex-3.52



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.51) | Index | Next exercise (3.53) >>

perry

```
(define seq (stream-map accum (stream-enumerate-interval 1 20)))  
  
sum  
;Value: 1  
  
(define y (stream-filter even? seq))  
  
sum  
;Value: 6  
  
(define z (stream-filter (lambda (x) (= (remainder x 5) 0)) seq))  
  
(stream-ref y 7)  
;Value: 136
```

leafac

```
(define sum 0)  
;; sum => 0  
  
(define (accum x)  
  (set! sum (+ x sum))  
  sum)  
;; sum => 0  
  
(define seq (stream-map accum (stream-enumerate-interval 1 20)))  
;; sum => 1  
  
(define y (stream-filter even? seq))  
;; sum => 6  
  
(define z (stream-filter (lambda (x) (= (remainder x 5) 0))  
                         seq))  
;; sum => 10  
  
(stream-ref y 7)  
;; sum => 136  
;; => 136  
  
(display-stream z)  
;; sum => 210  
;; => (10 15 45 55 105 120 190 210)
```

If we had not memoized the results of delayed evaluations, we would need to recalculate the elements that `stream-ref` created when we were evaluating `display-stream`. Thus, the results would be different, because the accumulator would add those to the sum twice.

timothy235

Note that if a stream has side-effects, the side-effect of the stream-car will only ever be expressed once, and this is true for memoized and un-memoized streams. This is because of our definition of a stream as a value-thunk pair. The definition of stream in effect automatically caches the stream-car as the value part of the value-thunk pair.

krubar

This is not true. See codybartfast answer.

codybartfast

	SICP Scheme With Memoization	SICP Scheme Without Memoization	Racket With Text's Map & Filter	Racket With Built in Map & Filter
sum after:				
define accum	0	0	0	0
define seq	1	1	0	0
define y	6	6	6	0
define z	10	15	10	0
stream-ref	136	162	136	136
display-stream	210	362	210	210

Printed response with memoization: 10, 15, 45, 55, 105, 120, 190, 210

Printed response without memoization: 15, 180, 230, 305

Printed response with Racket: 10, 15, 45, 55, 105, 120, 190, 210

Unlike generators and streams from most languages, including modern Scheme, the first element of the stream is not delayed and is evaluated at creation time. With memoization this doesn't make a difference to the elements of seq (and hence values of sum) as they are evaluated just once and always in the same order. But without memoization several elements are evaluated more than once and the order in which they are evaluated will affect the values of seq.

The lack of a delay for the first item of a stream is discussed in the Rationale of SRFI 41 (Scheme Request For Implementation) where Abelson and Sussman's implementation is described as 'odd' streams and this chapter of SICP is referenced for understanding 'odd' streams. However, 'even' streams (Wadler et al), which do delay the first item, predominate today.

The non-memoizing results were obtained by implementing a non-memoizing stream using the language implementation from Chapter 4.

```
=====
== With Memoization =====
=====

Call to define seq:
=====
    sum: 0 |
    interval: | 1
    -----
    seq: | 1

Call to define y:
=====
    sum: 1 |
    car seq: 1 | 1
    interval: 1 | 2 3
    seq: | 3 6
    -----
    y: | - - 6

Call to define z:
=====
    sum: 6 |
    car seq: 1 | 1
    car y: 6 |
    interval: 3 | 4
    seq: | 10
    memoized: | 3 6
    -----
    z: | - - - 10

Call to stream-ref y 7
=====
    sum: 10 |
    car seq: 1 |
    car y: 6 | 6
    car z: 10 |
    interval: 4 | 5 6 7 8 9 10 11 12 13 14 15 16
    seq: | 15 21 28 36 45 55 66 78 91 105 120 136
    memoized: | 10
```

```

-----+
stream-ref: | 6 10 - - 28 36 - - 66 78 - - 120 136

Call to display-stream
=====
    sum: 136 |
    car seq: 1 |
    car y: 6 |
    car z: 10 |
interval: 16 |                               17 18 19 20
    seq:   | 153 171 190 210
memoized: | 15 21 28 36 45 55 66 78 91 105 120 136
-----+
display-stream:| 10 15 - - - 45 55 - - - 105 120 - - - 190 210

=====
== Without Memoization ==
=====

Call to define seq:
=====
    sum: 0 |
interval: | 1
-----+
    seq: | 1

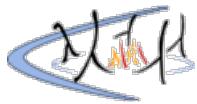
Call to define y:
=====
    sum: 1 |
    car seq: 1 | 1
interval: 1 | 2 3
    seq: | 3 6
-----+
    y: | - - 6

Call to define z:
=====
    sum: 6 |
    car seq: 1 | 1
    car y: 6 |
interval: 1 | 2 3 4
    seq: | 8 11 15
-----+
    z: | - - - 15

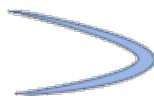
Call to stream-ref y 7
=====
    sum: 15 |
    car seq: 1 |
    car y: 6 | 6
    car z: 15 |
interval: 3 | 4 5 6 7 8 9 10 11 12 13 14 15 16 17
    seq: | 19 24 30 37 45 54 64 75 87 100 114 129 145 162
-----+
streamm-ref: | 6 - 24 30 - - 54 64 - - 100 114 - - 162

Call to display-stream
=====
    sum: 162 |
    car seq: 1 |
    car y: 6 |
    car z: 15 | 15
interval: 4 | 5 6 7 8 9 10 11 12 ... 16 17 18 19 20
    seq: | 167 173 180 188 197 207 218 230 ... 288 305 323 342 362
-----+
display-stream:| 15 - - 180 - - - 230 ... - 305 - - -

```



# sicp-ex-3.53



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.52) | Index | Next exercise (3.54) >>

*;; s produces a stream of powers of 2, akin to the double stream example.*

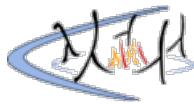
```
Nue
(define s (cons-stream 1 (add-streams s s)))

;; this is the same as
(define s (cons-stream 1 (stream-scale 2 s)))

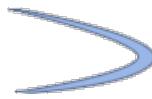
;; which is the same as
(define s (cons-stream 1 (stream-scale 2 (cons-stream 1 (stream-scale 2 s))))))

;; we're scaling by 2 repeatedly. its the powers of 2
```

Last modified : 2021-05-15 04:59:38  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-3.54



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.53) | Index | Next exercise (3.55) >>

```
(define factorials (cons-stream 1 (mul-streams (add-streams ones integers) factorials)))
```

leafac

I believe the above is wrong, because `integers' start at 1 and we want the factorial of `n + 1'. Also, it's missing the `mul-streams'. Here's my solution:

```
(define (mul-streams s1 s2)
  (stream-map * s1 s2))

(define factorials (cons-stream 1 (mul-streams integers factorials)))
```

ict

To make the stream have the nth element be n+1!, shoudn't the function be:

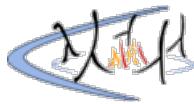
```
(define factorials
  (cons-stream 1
              (mul-streams factorials (stream-cdr integers))))
```

Shawn

I agree with you. I also have the same definition.

Last modified : 2016-08-06 16:22:05  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-3.55



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.54) | Index | Next exercise (3.56) >>

meteorgan  

```
(define (partial-sums s)
      (cons-stream (stream-car s) (add-streams (stream-cdr s) (partial-sums
      s))))
```

dzy this will cause recalculation.

hunzhan  

```
(define (partial-sums s)
      (add-streams s (cons-stream 0 (partial-sums s))))
```

Mathieu Borderé @hunzhan: nice, very elegant

lertecc Must have self-reference to avoid recalculation:  

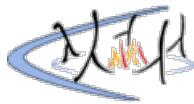
```
(define (partial-sums s)
  (define ps (add-streams s (cons-stream 0 ps)))
  ps)
```

dekuofa1995 My solution:

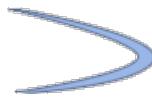
```
(define (partial-sums S)
  (cons-stream (stream-car S) (add-streams (partial-sums (stream-cdr S)) S)))
```

j-minster dekuofa1995's solution is wrong. For his, (partial-sums integers) returns 1, 3, 7, 12... correct solution should return (partial-sums integers) => 1, 3, 6, 10...

```
(define (partial-sums s)
  (cons-stream (stream-car s)
    (add-streams (partial-sums s)
      (stream-cdr s))))
```



# sicp-ex-3.56



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.55) | Index | Next exercise (3.57) >>

meteorgan

```
(define S (cons-stream 1 (merge (merge (scale-stream S 2) (scale-stream S 3))  
                                (scale-stream S 5)))))
```

master

Is there a difference between these two definitions?

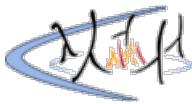
```
(define S (cons-stream 1 (merge (scale-stream S 2) (merge (scale-stream S  
3) (scale-stream S 5))))
```

For some reason it feels more symmetrical to put most of the weight so to speak at the end, I don't think it changes anything but I want to make sure. Unfortunately Racket doesn't seem to allow recursive datatypes like this so there isn't any way to test. I would like to work through this book using MIT Scheme but Gentoo doesn't have it in the repos for some reason and the source files from gnu.org don't compile for me. Can anybody using MIT Scheme confirm?

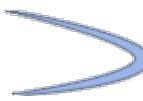
squarebat

This will work correctly. If you can install drracket in gentoo then you can use MIT scheme with the #lang sicp package

Last modified : 2021-09-25 14:32:29  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-3.57



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.56) | Index | Next exercise (3.58) >>

adams

Every term in fibs is computed first as the sum of two previous terms. Since fibs is memoized recalling each previous term requires no additions. Thus each term after the first uses one addition. To compute the nth term (starting at n=0), max(0, n-1) additions are made. If we had not memoized fibs, then to recall previous terms in fibs would require each term to be recomputed. Since the only terms not computed are 1 and zero, fib(n) would require at least fib(n) - 1 additions which is exponential.

xdaividliu

This exercise is referring to the implementation of fibs in the book which looks like this:

```
(define fibs
  (cons-stream 0
    (cons-stream 1
      (add-streams (stream-cdr fibs)
        fibs))))
```

The call to add-streams must read the previous two elements of fibs. When delay is memoized, reading the previous two elements can be done with a lookup, without any additions, and the only addition is the one performed immediately after that. Every adjacent pair of Fibonacci numbers are added only once; the first time. Hence, the total number of additions to get the n-th Fibonacci number is  $O(n)$ .

If delay were \*not\* memoized, then the lookup part of the previous two elements would be replaced by a full computation, since every element in a stream besides the very first one is delayed and must be forced every time it is accessed. Hence, "looking up" the values of the two previous Fibonacci numbers now requires all the additions needed to compute those two numbers in the first place.

Let's define  $A(n)$  as the number of additions required to obtain the n-th Fibonacci number, assuming that delay is implemented \*without\* memoization. Then, we have  $A(n) = A(n-1) + A(n-2)$ , which is exactly the Fibonacci recursion relation. This recursion has the solution  $A = O(\phi^n)$  where  $\phi$  is the golden ratio. Hence, the number of additions when delay is not memoized is exponential.

As an aside, the book also presents an alternative implementation of fibs:

```
(define (fibgen a b)
  (cons-stream a (fibgen b (+ a b)))))

(define fibs (fibgen 0 1))
```

This implementation uses an iterative model, not a recursive one, so it takes  $O(n)$  additions even when delay is not memoized.

Brandon

Agree with your conclusion: "This recursion has the solution  $A = O(\phi^n)$  where  $\phi$  is the golden ratio. Hence, the number of additions when delay is not memoized is exponential."

I have a question with how you arrived at this. You stated that the number of additions required to obtain the n-th Fibonacci number is  $A(n) = A(n-1) + A(n-2)$ .

However, because this is a stream, don't we also need to compute the Fibonacci numbers for the range  $n=[0,n-1]$ . Example, when computing Fib(3):

- Compute Fib(0)
- Compute Fib(1)
- Compute Fib(2) by computing Fib(0) and Fib(1) and adding
- Compute Fib(3) by computing Fib(2) [which requires computing Fib(0) and Fib(1) again] and Fib(1), and adding

If this is true, this would yield the number of additions as:

```
A(n) = A(n-1) + [A(n-1) + A(n-2) + 1]
```

Sphinxsky

```
; First, you can answer the first question after modifying the add-streams
(define counter 0)

(define (add-streams stream1 stream2)
  (stream-map
    (lambda (a b)
      (begin
        (set! counter (1+ counter))
        (+ a b)))
    stream1
    stream2))

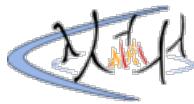
;; let count-add(fib(n)) = c(n)
;; then c(n+2) = c(n+1) + c(n) + 1
;; so you can get the c(n) like this
(define (count-add n)
  (+
    (* (/ (+ 5 (sqrt 5)) 10)
        (expt (/ (+ 1 (sqrt 5)) 2) n))
    (* (/ (- 5 (sqrt 5)) 10)
        (expt (/ (- 1 (sqrt 5)) 2) n)))
    (- 1)))

;; so Θ(c(n)) = Θ(φ^n)
```

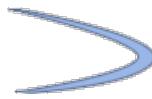
Cirno

```
; with memo
; k = 0, no force
; k = 1, no force
; k = 2, 1 force
; k = 3, 1 force
; k = 4, 1 force
; .
; .
; .
; k = n, 1 force
; total = (n-2) forces

; no memo
; for each force, we have to add up all the previous forces for fib and for
; (stream-cdr fib), so we get fib(k-1) + fib(k-2), but that's exactly fib(k).
; k = 0, no force
; k = 1, no force
; k = 2, 1 force
; k = 3, 2 force
; k = 4, 3 force
; k = 5, 5 force
; k = 6, 8 force
; .
; .
; .
; k = n, fib(n) force
; total = fib(n+2)-1-1, which is equal to the sum of the first n numbers with
; fib(1) excluded, but that's equal to (phi^(n+2)-psi^(n+2))/sqrt(5) - 2
```



# sicp-ex-3.58



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.57) | Index | Next exercise (3.59) >>

meteorgan

```
The result is the rational value that num divide den in base radix.  
(expand 1 7 10)  
=> 1 4 2 8 5 7 4 2 8 5 7 ...  
(expand 3 8 10)  
=> 3 7 5 0 0 0 ...
```

danhuynh

I think @meteorgan miscalculate (expand 1 7 10). => 1 4 2 8 5 7 1 4 2 8 5 7 ...

j-minster

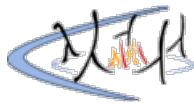
edit: I noticed meteorgan's sequence is missing a '1', so  
danhuynh is correct.

```
(map (lambda (s) (stream-ref (expand-rad 1 7 10) s))  
  '(0 1 2 3 4 5 6 7 8 9 10))  
  
;=> (1 4 2 8 5 7 1 4 2 8 5)
```

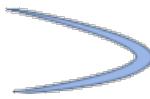
xdaividliu

The result is the floating-point representation of (/ num den) with radix as the base.

```
; list of first n elements of stream  
(define (partial-stream->list stream n)  
  (define (rec str i)  
    (if (= i n)  
        ()  
        (cons (stream-car str)  
              (rec (stream-cdr str) (1+ i))))))  
  (rec stream 0))  
  
(partial-stream->list (expand 1 7 10) 10)  
;Value: (1 4 2 8 5 7 1 4 2 8)  
  
(/ 1.0 7)  
;Value: .14285714285714285  
  
(partial-stream->list (expand 3 8 10) 5)  
;Value: (3 7 5 0 0)  
  
(/ 3.0 8)  
;Value: .375
```



# sicp-ex-3.59



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.58) | Index | Next exercise (3.60) >>

meteorgan

```
a)
(define (integrate-series s)
  (stream-map * (stream-map / ones integers) s))
b)
(define sine-series (cons-stream 0 (integrate-series cosine-series)))
(define cosine-series (cons-stream 1 (integrate-series (scale-stream sine-series -1))))
```

shawn

```
a)
(define (integrate-series s)
  (stream-map / s integers))

Here's a simpler version of a).
```

Sphinxsky

```
; ; a)
(define (integrate-series stream)
  (stream-map / stream integers))
; ; b)
(define cosine-series
  (cons-stream 1 (stream-map - (integrate-series sine-series))))
(define sine-series
  (cons-stream 0 (integrate-series cosine-series)))

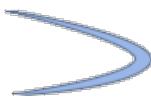
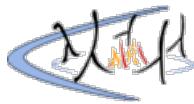
; ; other way to exp
(define exp-series
  (stream-map / ones (cons-stream 1 factorials)))
```

zz

```
; (a)
(define (integrate-series s)
  (stream-map / s integers))
(define exp-series (cons-stream 1
  (integrate-series exp-series)))
; (b)
(define (filter-odd s) (cons-stream (stream-car s) (filter-odd (stream-cdr (stream-cdr s)))))
(define (filter-even s) (cons-stream (stream-car (stream-cdr s)) (filter-even
(stream-cdr (stream-cdr s)))))
(define s-11 (cons-stream 1 (stream-map (lambda (e)
  (if (= 1 e)
    -1
    1)) s-11)))
(define cosine-series (stream-map * s-11
  (filter-odd exp-series)))
(define sine-series (stream-map * s-11
  (filter-even exp-series)))
```



# sicp-ex-3.60



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.59) | Index | Next exercise (3.61) >>

meteorgan

```
(define (mul-series s1 s2)
  (cons-stream (* (stream-car s1) (stream-car s2))
              (add-streams (scale-stream (stream-cdr s2) (stream-car s1))
                           (mul-series (stream-cdr s1) s2))))
```

zzd3zzd

```
(define (mul-series s1 s2)
  (cons-stream (* (stream-car s1)
                  (stream-car s2))
              (add-streams (mul-streams (stream-cdr s1)
                                         (stream-cdr s2))
                           (mul-series s1 s2)))

(define (mul-series2 s1 s2)
  (partial-sums (mul-streams s1 s2)))
```

atupal

```
(define (mul-series s1 s2)
  (cons-stream
    (* (stream-car s1) (stream-car s2))
    (add-streams (add-streams (scale-stream (stream-cdr s1) (stream-car s2))
                               (scale-stream (stream-cdr s2) (stream-car s1)))
                 (cons-stream 0 (mul-series (stream-cdr s1) (stream-cdr
s2)))))))
```

jeff

atupal is correct

seek

meteorgan and atupal are correct and zzd3zzd is wrong according to my checker.

```
(define (display-stream-until n s)      ; n-th value included
  (if (< n 0)
      the-empty-stream
      (begin (newline) (display (stream-car s))
             (display-stream-until (- n 1) (stream-cdr s)))))

; It can be easily seen that (mul-series (stream-cdr s1) s2) equals
; (add-streams (scale-stream (stream-cdr s2) (stream-car s1))
;              (cons-stream 0 (mul-series (stream-cdr s1) (stream-cdr s2))))
; , which means that meteorgan's solution and atupal's solution are equivalent.
(define (mul-series s1 s2)
  (cons-stream (* (stream-car s1) (stream-car s2))
              (add-streams (scale-stream (stream-cdr s1) (stream-car s2))
                           (mul-series (stream-cdr s2) s1)))

(define circle-series
  (add-streams (mul-series cosine-series cosine-series)
               (mul-series sine-series sine-series)))
```

```
; if you see one 1 followed by 0's, your mul-series is correct.  
(display-stream-until 30 circle-series)
```

Patrick-OHara

I agree with seok. However, in testing with non-infinite series I found that the book's version of stream-map does not cope with series of different lengths, and so gives an incorrect result for mul-series applied to non-infinite series. See <http://community.schemewiki.org/?sicp-ex-3.50> for my fix for this.

joshroybal

```
(define (mul-series s1 s2)  
  (cons-stream (* (stream-car s1) (stream-car s2))  
              (add-streams  
                (scale-stream (stream-cdr s2) (stream-car s1))  
                (mul-series (stream-cdr s1) s2))))
```

---

Last modified : 2022-05-14 01:41:51  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-3.61

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.60) | Index | Next exercise (3.62) >>

meteorgan

```
(define (reciprocal-series s)
  (cons-stream 1 (scale-stream (mul-series (stream-cdr s) (reciprocal-series s)) -1)))
```

leafac

I'm not sure the above solution works, because calling (reciprocal-series s) on the body creates a new stream, which results are not memoized.

Here's my version:

```
(define (invert-unit-series series)
  (define inverted-unit-series
    (cons-stream 1 (scale-stream (mul-streams (stream-cdr series)
                                              inverted-unit-series)
                                 -1)))
  inverted-unit-series)
```

lockywolf

Why is it mul-streams, not mul-series?

xdaavidliu

The point that leafac makes is exactly the topic of exercise 3.63; the very next exercise.

Last modified : 2020-01-04 03:36:09  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-3.62

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.61) | Index | Next exercise (3.63) >>

cyzx

A simple solution, that produces the right tangent power series coefficients when displayed

```
;nums is numerator, dems is denominator

(define (div-series nums dems)
  (mul-series nums
    (invert-series dems)))

(define tangent-series (div-series sine-series cosine-series))
```

Source for the correct tangent power series: <https://socratic.org/questions/what-is-the-taylor-series-expansion-for-the-tangent-function-tanx>

meteorgan

```
(define (div-series s1 s2)
  (let ((c (stream-car s2)))
    (if (= c 0)
        (error "constant term of s2 can't be 0!")
        (scale-stream (mul-series s1
          (reciprocal-series (scale-stream s2 (/ 1 c))))
          (/ 1 c)))))

(define tan-series (div-series sine-series cosine-series))
```

anuj

I'm pretty sure the tan series won't work like that because half the coefficients for cos are zero.

gravitykey

```
; according 3-61
; S·X = 1
; X = 1 / S
; another S1
; S1 / S = S1 · X

;use code in 3-61
(load "3-61.scm")
;"daoshu" mean reciprocal

(define (div-series s1 s2)
  (cond ((eq? 0 (stream-car s2)) (error "constant term of s2 can't be 0!"))
        (else (mul-series s1 (daoshu s2)))))
```

tests

```
(define sine-series (cons-stream 0 (integrate-series cosine-series)))
(define cosine-series (cons-stream 1 (integrate-series (stream-map (lambda (x) (* x -1))
  sine-series))))
(define (add-streams s1 s2) (stream-map + s1 s2))

(define (integrate-series stm)
  (define (inner s a)
    (cons-stream (* (/ 1 a) (stream-car s))
      (inner (stream-cdr s) (+ a 1))))
  (inner stm 1))

(define tan-series (div-series sine-series cosine-series))
(stream-head tan-series 10)
```

```
;Value 13: (0 1 0 1/3 0 2/15 0 17/315 0 62/2835)
;It's correct.
```

Sphinxsky

```
(define (div-series stream1 stream2)
  (let ((s2-car (stream-car stream2)))
    (if (= s2-car 0)
        (error "Denominator constant term cannot be 0 ! -- DIV-SERIES" s2-car)
        (let ((reciprocal-s2-car (/ 1 s2-car)))
          (mul-series
            (scale-stream stream1 reciprocal-s2-car)
            (reciprocal-series (scale-stream stream2 reciprocal-s2-car)))))))

(define tangent-series (div-series sine-series cosine-series))
```

---

Last modified : 2020-08-19 10:48:26  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-3.63

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.62) | Index | Next exercise (3.64) >>

leafac

The function defined by Louis calls `(sqrt-stream x)' recursively, which yields a new sequence with the same values, only not memoized yet. Thus this version is slower, but if memoization didn't take place, both implementations would be the same.

cel

```
(define (louis-sqrt-stream x)
  (cons-stream 1.0
    (stream-map (lambda (guess) (sqrt-improve guess x))
      (louis-sqrt-stream x))))
```

To access the second element in the stream `(louis-sqrt-stream n)' by taking the stream-car of `(stream-cdr (louis-sqrt-stream n))' requires two evaluations of `(louis-sqrt-stream n)': one as the argument to `stream-cdr' and one in the course of applying that procedure to its argument. Thus one redundant evaluation is performed. Similarly, accessing the third element in the stream by taking the stream-car of `(stream-cdr (stream-cdr (louis-sqrt-stream n)))' requires two evaluations of `(stream-cdr (louis-sqrt-stream n))': one as the argument to `stream-cdr' and one in the course of applying that procedure to its argument. (Memoization is irrelevant here, since the interpreter cannot see that the second application of `stream-cdr' to `(louis-sqrt-stream n)' is an application to the "same" object as the first application, due to the way Louis designed his procedure.) Thus one redundant evaluation is performed, on top of the redundancies involved in evaluating `(stream-cdr (louis-sqrt-stream n))' twice. In general, to access the  $(k + 1)$ th element in the stream (by  $k + 1$  applications of `stream-cdr' followed by one application of `stream-car') requires one more than twice the number of redundant evaluations required to access the  $k$ th element. In other words, the number of redundant evaluations grows exponentially with the index of the element accessed. As already hinted, there would not have been further redundancy if our implementation of delay did not use memoization. For as a result of the distinctness of an "external" call to `(louis-sqrt-stream n)' from its own internal call to `(louis-sqrt-stream n)', the two calls to `(stream-cdr (louis-sqrt-stream n))' made in the course of evaluating `(stream-cdr (stream-cdr (louis-sqrt-stream n)))', for example, cannot exploit memoization because the interpreter cannot see that `stream-cdr' is being applied to the "same" object in each case.

```
(define (sqrt-stream x)
  (define guesses
    (cons-stream 1.0
      (stream-map (lambda (guess) (sqrt-improve guess x))
        guesses)))
  guesses)
```

Unlike the `louis-sqrt-stream' procedure, accessing the second element in the stream `(sqrt-stream n)' by taking the stream-car of `(stream-cdr (sqrt-stream n))' does not create an second, internal call to `(sqrt-stream n)'. Instead, evaluation of the `(sqrt-stream n)' argument passed to `stream-cdr' merely assigns the `guesses' variable to a value in the environment. This fact prevents the exponential growth of redundant evaluations observed in Louis's procedure. If, however, our implementation of delay did not use memoization, redundant evaluations would arise in another way. For example, to access the third element in the stream by taking the steam-car of `(stream-cdr (stream-cdr (sqrt-stream n)))' would require two "full" evaluations of `(stream-cdr guesses)': one as (a reduction of) the argument passed to `stream-cdr' and, due to the absence of memoization, another one in the course of evaluating the application of that procedure to its argument. In general, without the memoization optimization of delay, accessing the  $(k + 2)$ th element in the stream would require one more than twice the number of redundant evaluations required to access the  $(k + 1)$ th element. In other words, the number of redundant evaluations would grow exponentially with the index of the element accessed -- in the same way that Louis's procedure already does notwithstanding memoization.

seek

The complexity of `louis-sqrt-stream' is  $O(n^2)$ , not  $O(e^n)$  like cel said.

Here I denote `(lambda (guess) (sqrt-improve guess x))' as `sqrt-improve-x' and abbreviate `cons-stream' and `stream-map' as `cons' and `map' for the sake of simplicity.

```
;; Accessing 1st element
(cons 1.0
  (map sqrt-improve-x
    (louis-sqrt-stream x)))
```

```

;; Accessing 2nd element
(cons 1.0
      (map sqrt-improve-x
           (cons 1.0 ;; it has to go through one `map` to reach the top-most stream
                 (map sqrt-improve-x
                      (louis-sqrt-stream x)))))

;; Accessing 3rd element
(cons 1.0
      (map sqrt-improve-x
           (cons 1.0
                 (map sqrt-improve-x
                      (cons 1.0 ;; it has to go through two `map`s
                            (map sqrt-improve-x
                                 (louis-sqrt-stream x)))))))

;; Accessing 4th element
(cons 1.0
      (map sqrt-improve-x
           (cons 1.0
                 (map sqrt-improve-x
                      (cons 1.0
                            (map sqrt-improve-x
                                (cons 1.0 ;; it has to go through three `map`s
                                      (map sqrt-improve-x
                                         (louis-sqrt-stream x)))))))

;; ...

```

It's easy to see that accessing n-th element consists of  $(n-1)$  applications of `stream-cdr`, one expansion of `louis-sqrt-stream` and  $(n-1)$  applications of `stream-map` , which means that the complexity is  $O(n^2)$ .

It can be checked empirically with the following procedure:

```

(define (sqrt-stream x)
  (cons-stream 1.0 (stream-map (lambda (guess) (sqrt-improve guess x))
                               (sqrt-stream x))))
(define (sqrt-improve guess x)
  (average guess (/ x guess)))
(define (average x . xs)
  (/ (apply + (cons x xs)) (+ (length xs) 1)))

(define (time proc . args)
  (newline)
  (display "time-elapsed: ")
  (let ((start-time (runtime)))
    (define res (apply proc args))
    (display (- (runtime) start-time))
    res))

; warm-up
(define sqrt2 (sqrt-stream 2))
(stream-ref sqrt2 4000)

; test
(define sqrt2 (sqrt-stream 2))
(time stream-ref sqrt2 1000)
; 0.48

(define sqrt2 (sqrt-stream 2))
(time stream-ref sqrt2 2000)
; 1.85 ~= 0.48 * 3.9

(define sqrt2 (sqrt-stream 2))
(time stream-ref sqrt2 3000)
; 4.18 ~= 0.48 * 8.7

(define sqrt2 (sqrt-stream 2))
(time stream-ref sqrt2 4000)
; 7.45 ~= 0.48 * 15.5

```

# sicp-ex-3.64

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.63) | Index | Next exercise (3.65) >>

meteorgan

```
(define (stream-limit stream tolerance)
  (if (< (abs (- (stream-ref stream 1) (stream-ref stream 0))) tolerance)
      (stream-ref stream 1)
      (stream-limit (stream-cdr stream) tolerance)))
```

luckykoala

```
(define (stream-limit stream tolerance)
  (define (good-enough? a b)
    (< (abs (- a b)) tolerance))
  (define (loop s)
    (let ((a (stream-ref s 0))
          (b (stream-ref s 1)))
      (if (good-enough? a b)
          b
          (loop (stream-cdr s)))))
  (loop stream)))
```

Sphinxsky

```
(define (sub-streams stream1 stream2)
  (stream-map - stream1 stream2))

(define (one-order-difference stream)
  (sub-streams (stream-cdr stream) stream))

(define (stream-limit stream tolerance)
  (define (iter tolerance-stream stream)
    (if (<= (abs (stream-car tolerance-stream)) tolerance)
        (stream-car stream)
        (iter (stream-cdr tolerance-stream) (stream-cdr stream))))
  (iter (one-order-difference stream) (stream-cdr stream)))
```

# sicp-ex-3.65

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.64) | Index | Next exercise (3.66) >>

meteorgan

```
(define (ln2-summands n)
  (cons-stream (/ 1.0 n)
               (stream-map - (ln2-summands (+ n 1)))))

(define ln2-stream
  (partial-sums (ln2-summands 1)))
```

davl

Just wondering why not using recurrence relation to define summands, for instance for ln:

```
(define ln-summands
  (cons-stream 1
    (stream-map (lambda (x)
      (* (if (> x 0) -1 1) (/ 1 (+ (denominator x) 1)))
      ln-summands)))

(define ln-stream (stream-map exact->inexact (partial-sums ln-summands)))
```

Sphinxsky

```
(define (sub-streams stream1 stream2)
  (stream-map - stream1 stream2))

(define (one-order-difference stream)
  (sub-streams (stream-cdr stream) stream))

(define (euler-transform s)
  (sub-streams
    (stream-cdr (stream-cdr s))
    (stream-map
      /
      (stream-map square (one-order-difference (stream-cdr s)))
      (one-order-difference (one-order-difference s)))))

(define (ln2-summands n)
  (cons-stream
    (/ 1.0 n)
    (stream-map - (ln2-summands (+ n 1)))))

(define ln2-stream
  (partial-sums (ln2-summands 1)))

(define (show-streams n . streams)
  (if (accumulate
        (lambda (a b) (or a b))
        #f
        (cons (= n 0) (map stream-null? streams)))
    (newline)
    (begin
      (display-line (map stream-car streams))
      (apply show-streams (cons (- n 1) (map stream-cdr streams))))))

(define (make-ln2-one-order-difference-tableau)
  (let ((ln2-tableau (make-tableau euler-transform ln2-stream)))
    (lambda (i) (one-order-difference (stream-ref ln2-tableau i)))))

(define ln2-oodt (make-ln2-one-order-difference-tableau))

(show-streams 12 (ln2-oodt 4) (ln2-oodt 5) (ln2-oodt 6))
; It can be seen that the convergence rate of this series increases by 2-3 orders of
magnitude every time it passes through Euler transformation
```



# sicp-ex-3.66

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.65) | Index | Next exercise (3.67) >>

meteorgan

```
(1, 100) : 198
(100, 100): 99*2^200 - 1 + 2^99.
the equation for (m, n) is:
n = 1: 2^m - 1;
n = 2: 2^m - 1 + 2^(m-1);
n > 2: 2^m - 1 + 2^(m+1) + (n-2)*2^m
```

zzd3zzd

```
@meteorgan --- You are Wrong !!!
(1,100):198;
(100,100):2^100 - 1;
-----
f(n,m) m>=n (m,n is Z+)
(m-n=0): 2^n - 1
(m-n=1): (2^n - 1) + 2^(n - 1)
(m-n>1): (2^n - 1) + 2^(n - 1) + (m - n - 1) * 2^n
-----
1   2   3   4   5   6   7   8   9   ... 100
1 1   2   4   6   8   10  12  14  16   198
2   3   5   9   13  17  21  25  29
3       7   11  19  27  35  43  51
4           15  23  39  .....
5               31  .....
.
.
100 ----- (2^100 - 1)
```

draeklae

I can confirm zzd3zzd's formulas, but we should throw a -1 in (since the exercises asks for how many pairs \*precede\* (1,100) etc.). I add a somewhat simplified definition for f as a bonus:

```
f(i,j) = 2^i - 2, i = j
f(i,j) = 2^i * (j-i) + 2^(i-1) - 2, i < j
```

luckykoala

I wrote this in Scheme:

```
(define (index-of-pair pair)
  (let ((i (car pair))
        (j (cadr pair)))
    (cond ((> i j) #f)
          ((= i j) (+ (expt 2 i) -1))
          (else (+ (* (expt 2 i) (- j i))
                    (expt 2 (- i 1))
                    -1))))
```

mart256

Thanks @zzd3zzd for the diagram. It was very useful to understand the pattern. Next time I'll try to figure patterns with diagrams like this.

xdaividliu

Let (z m n) be the 0-based index of the pair (m n) in the stream (pairs integers integers). In other words, we want to find z such that

```
(equal? (list m n)
       (stream-ref (pairs integers integers) (z m n)))
```

For  $m = 1$ , by a bit of simple inspection of the first few elements of the stream, we have

```
(= (z 1 n)
  (* 2 (- n 3)))
```

For  $m = 2$ , by some further inspection, we have

```
(= (z 2 n)
  (+ 2
    (* 2
      (z 1 (-1+ n)))))
```

Since the streams and interleaved sub-streams are self-similar, this relation holds for all  $m$ .

```
(= (z m n)
  (+ 2
    (* 2
      (z (-1+ m) (-1+ n)))))
```

This recursion can be solved straightforwardly with pencil and paper to obtain

```
(define (z m n)
  (+ -2
    (* (expt 2 m)
      (+ n 1/2 (- m)))))

;; for example:
(z 6 14) ; 542
(stream-ref (pairs integers integers) 542) ; (6 14)
```

newone

This is how I confirm draeklae's answer.

For an integer pair  $(m, n)$ , the number of its predecessors is given by

$$f(m,n) = 2^{\{m\}} - 2, \text{ for } m=n,$$

$$f(m,n) = (n-m)*2^{\{m\}} + 2^{\{m-1\}} - 2, \text{ for } m < n.$$

Let's just take a look at the index  $(i,j)$  starting from  $(0,0)$  since it's easy to substitute  $(i,j)$  for the integer pair  $(m,n)$  with  $m=i+1$  and  $n=j+1$ .

(0,0)	(0,1) (0,2) (0,3) ...
-----	
(1,1) (1,2) (1,3) ...	
(2,2) (2,3) ...	

The triangular matrix have three parts:

I.  $(0,0)$ , the leading pair.

II.  $(0,1) (0,2) (0,3) \dots$ , the rest of the first row.

III.  $(1,1) (1,2) (1,3) \dots$ , except the first row.

This structure remains whenever we focus on a specified row. For the  $i$ th row, the procedure `interleave` always interleaves part II with part III. It's clear that `interleave` will double the distance between adjoint pairs. That means, interleaving the part II of the  $i$ th row with the rows after it will scale the distance between  $(i,i+1)$  and  $(i,j)$  from 1 to 2 for any  $j > i+1$ .

After collapsing rows after the  $i$ th row, the coordinate of  $(i,j)$  relative to  $(i,i)$  is given by:

```
0,          for i=j,
2(j-i)-1, for i<j.

... rows precede the ith row ...

(i,i) | (i,i+1) (i,i+2) (i,i+3) ...
-----|
| ... rows after ...
| ... the ith row ...
| interleaving these pairs
| doubles the distance
| between (i,j) and (i,j+1)
| for any j>0
After collapsing =>
... rows precede the ith row ...
```

```

----- (i,i) | (i,i+1) (?,?) (i,i+2) (?,?) (i,i+3) ...
D:   --1--- -----2----- -----2-----

```

Interleaving the collapsed ith row with the part II of the (i-1)th row will double the distance between (i,i) and (i,j). If we do this i times, the distance will be doubled i items as well. The total distance from (i,i) to (i,j) where  $i < j$  is hence  $(2j-2i-1)*2^{\{i\}}$ .

What left is to figure out the distance from (0,0) to (i,i).

If we interleave the collapsed ith row with the (i-1)th row recursively, the pair (i,i) will shift to the right with a pattern:

```

- 0, counter=i
- 2, counter=i-1
- 2*2+2, counter=i-2
...
- 2^{\{i\}}+2^{\{i-1\}}+...+2, counter=0

```

This pattern gives the distance from (0,0) to (i,i),

$2^{\{i+1\}}-2$ .

Adding up the distance from (0,0) to (i,i) and the distance from (i,i) to (i,j) gives

$(j-i)*2^{\{i+1\}}+2^{\{i\}}-2$ .

Here we get the formulas

$f(i,j)=2^{\{i+1\}}-2$ , for  $i=j$ ,

$f(i,j)=(j-i)*2^{\{i+1\}}+2^{\{i\}}-2$ , for  $i < j$ .

Substituting (i,j) for (m-1,n-1) should give the count of integer pairs before (m,n).

$f(m,n)=2^{\{m\}}-2$ , for  $m=n$ ,

$f(m,n)=(n-m)*2^{\{m\}}+2^{\{m-1\}}-2$ , for  $m < n$ .

```

(define (count-integer-pairs m n)
  (if (= m n)
      (- (expt 2 m) 2)
      (+ (* (- n m) (expt 2 m)) (expt 2 (- m 1)) -2)))

```

brandon

I arrived at a similar answer to @zzd3zzd.

Note that I used 0-based indexing. Thus, pair (1,1) is at position 0.

My equations:

```

k(i,n) is the position of pair (i,n).

if (i==n) : k(i,n)=(2^i) - 2
if (n==i+1) : k(i,n)=k(i,i) + 2^(i-1)
else : k(i,n)=k(i,i+1) + (n-i-1)*2^i

```

Therefore:

```

k(1,1) = 2^1 - 2 = 0
k(1,2) = k(1,1) + 2^0
          = 0 + 1 = 1

k(1,100) = k(1,2) + (100-1-1)*2^1
           = 1 + (98)*2
           = 197

k(99,99) = 2^99 - 2
k(99,100) = k(99,99) + 2^98
             = 2^99 - 2 + 2^98

k(100,100) = 2^100 - 2

```

How I arrived at this answer: TODO



# sicp-ex-3.67

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.66) | Index | Next exercise (3.68) >>

3pmtea

The table can still be divided into 3 parts:

- The 1st column of the 1st row
- The rest of the 1st row
- The rest rows

```
;;
(define (pairs s t)
  (cons-stream
    (list (stream-car s) (stream-car t))
    (interleave
      (stream-map (lambda (x) (list (stream-car s) x))
                  (stream-cdr t))
      (pairs (stream-cdr s) t))))
```

genovia

```
(define (pairs s t)
  (define (top ts tt)
    (cons-stream
      (list (stream-car ts) (stream-car tt))
      (interleave
        (stream-map (lambda (x) (list (stream-car ts) x))
                    (stream-cdr tt))
        (pairs (stream-cdr ts) (stream-cdr tt)))))

  (define (below bs bt)
    (cons-stream
      (list (stream-car bs) (stream-car bt))
      (interleave
        (stream-map (lambda (x) (list (stream-car bt) x))
                    (stream-cdr bs))
        (pairs (stream-cdr bs) (stream-cdr bt))))
    (interleave (top s t)
               (below (stream-cdr s) t))))
```

meteorgan

```
(define (all-pairs s t)
  (cons-stream
    (list (stream-car s) (stream-car t))
    (interleave
      (interleave
        (stream-map (lambda (x) (list (stream-car s)
                                       (stream-cdr t)))
                    (all-pairs (stream-cdr s) (stream-cdr t)))
        (stream-map (lambda (x) (list x (stream-car t)))
                    (stream-cdr s)))))))
```

mart256

@genovia I took a look at your answer and I think it would repeat pairs, I found the pair (1,3) repeated at least.

Here's my solution, I think it should work, it is similar to @meteorgan answer.

```
(define (pairs s t)
  (cons-stream
    (list (stream-car s) (stream-car t))
    (interleave
      (interleave
        (stream-map (lambda (x) (list (stream-car s)
                                       (stream-cdr t)))
                    (stream-map (lambda (x) (list x (stream-car t)))
                                (stream-cdr s)))))))
```

```
s)))  
          (stream-cdr t)))  
(pairs (stream-cdr s) (stream-cdr t))))
```

xdavidliu

Use the same logic as the table diagram in the book: the stream now consists of \*four\* parts, not three, the upper-left corner, the top row, a new "left column", and the lower-right block

```
(define (all-pairs s t)  
  (cons-stream  
    (list (stream-car s) (stream-car t)) ;; upper-left corner  
    (interleave  
      (interleave  
        (stream-map (lambda (x) (list (stream-car s) x))  
                    (stream-cdr t)) ;; top row  
        (stream-map (lambda (x) (list x (stream-car t)))  
                    (stream-cdr s))) ;; left column  
        (all-pairs (stream-cdr s) (stream-cdr t)))) ;; lower-right block  
  
(define (partial-stream->list stream n)  
  (define (rec str i)  
    (if (= i n)  
        ()  
        (cons (stream-car str)  
              (rec (stream-cdr str) (1+ i))))  
  (rec stream 0))  
;; utility function to return list of first n items  
  
(partial-stream->list (all-pairs integers integers) 10)  
;; ((1 1) (1 2) (2 2) (2 1) (2 3) (1 3) (3 3) (3 1) (3 2) (1 4))
```

yasser

My answer is similar to 3pmtea's answer. What do you guys think of it?

master

I'm also curious to know, I don't see any reason to make it so complicated.  
The start of the stream is  $(1,1)$ , after that an interleaving of  $(1,x)$  and  $((2,1)$   
and interleaving of  $(2,x)$  and  $((3,1)$  and interleaving of  $(3,x)$  etc ...

Won't such a stream contain all pairs?

krubar

Just by examining output of solution given by 3pmtea, it looks correct.

# sicp-ex-3.68

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.67) | Index | Next exercise (3.69) >>

meteorgan

no. The program will infinitely loop. because their is no delay in (pairs (stream-cdr s) (stream-cdr t)), it will be called recursively.

mart256

I disagree @meteorgan. `interleave` returns a cons-stream, then it has delay. Anyway, I think this solution won't work either, because it will produce the following sequence:

(1,1) (2,2) (1,2) (3,3) (1,3) (4,4) (1,4) (5,5) ...

The recursive call is always going to call to the second element of `interleave`, because there is no 'pivot' to do an interleaving, thus it leaves out elements like (3,4) (3,5) ... (4,5) (4,6) ... and so on.

anon

@meteorgan is correct here. `interleave` returns a cons-stream, but `interleave` will never be successfully called since its second arg (`pairs ...`) will be evaluated first. Attempting this evaluation will result in recursion sans base-case.

krubar

@anon, it is not true that second arg of `interleave` is evaluated first. In the (`interleave s1 s2`), car of `s1` is taken first then `interleave` calls itself with arguments changed around so that `s2` is now `s1` and (`stream-cdr s1`) becomes `s2`.

pvk

@anon is referring to the two arguments to `interleave`. You're correct that cons-stream (which appears in the definition of `interleave`) delays evaluation of its second argument, but (`interleave s1 s2`), like any ordinary procedure, must evaluate `s1` and `s2` first before it itself can be evaluated.

xdaividliu

Louis' implementation will recurse infinitely simply because `interleave` is an ordinary "function", not a special form like cons-stream, and hence will need to fully evaluate both arguments first, since Scheme uses eager evaluation for ordinary functions. Since the second argument to `interleave` is a recursive call to `pairs`, and there is no hard-coded base case, this implementation will recurse infinitely.

WLW

I agree with posts above about infinite loop. However, for me the `pairs` procedure also fails with finite streams such as (`stream-enumerate-interval 1 10`) - I get a contract violation error message - perhaps there is another problem with code too?

czyx

That is because (`stream-car s`) is continually called in the lambda function, despite the recursive call of `pairs` passing on (`stream-cdr s`) so at some point `s` would become empty in the case of a finite stream, but `stream-car` would still be called. Normally this error wouldn't pop up until the end of the stream is reached through successive `stream-cdr`'s, but this time as the entire stream is evaluate at the time of definition, it arises now itself.



# sicp-ex-3.69

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.68) | Index | Next exercise (3.70) >>

meteorgan + adams

I've made a small cleanup to meteorgan's code. Cons is much more appropriate than append. -- adams

```
(define (triples s t u)
  (cons-stream (list
                (stream-car s)
                (stream-car t)
                (stream-car u)))
    (interleave
      (stream-map (lambda (x) (cons (stream-car s) x))
                  (stream-cdr (pairs t u)))
      (triples (stream-cdr s)
                (stream-cdr t)
                (stream-cdr u)))))

(define (phythagorean-numbers)
  (define (square x) (* x x))
  (define numbers (triples integers integers integers))
  (stream-filter (lambda (x)
                  (= (square (caddr x))
                     (+ (square (car x)) (square (cadr x))))))
    numbers))
```

xdaividliu

The above solution requires many redundant computations of (pairs t u). Here's an alternative implementation that only computes (pairs t u) once. We construct the triples by consing all elements of s up to the corresponding first element in (pairs t u). For the case that t is not the integers, we compute (pairs integers integers) in order to keep track of how many elements of s "fit".

```
(define first-of-integer-pair
  (stream-map car (pairs integers integers)))

(define (triples s t u)
  (let ((pairs-tu (pairs t u))) ;* compute pairs only *once*
    (define (rec si i ptu top-i)
      (cons-stream
        (cons (stream-car si) (stream-car ptu))
        (if (= i (stream-car top-i))
            (rec s 1 (stream-cdr ptu) (stream-cdr top-i))
            ;* restart s cycle with next ptu
            (rec (stream-cdr si) (1+ i) ptu top-i)))
      (rec s 1 pairs-tu first-of-integer-pair)))

  ;* example: pythagorean triples

  (define triples-integers
    (triples integers integers integers))

  (define (pythagorean? a b c)
    (= (square c)
       (+ (square a) (square b)))))

  (define pythagorean-triples
    (stream-filter
      (lambda (triple)
        (apply pythagorean? triple))
      triples-integers))

  (stream-ref pythagorean-triples 0) ; (3 4 5)
  (stream-ref pythagorean-triples 1) ; (6 8 10)
  (stream-ref pythagorean-triples 2) ; (5 12 13)
  (stream-ref pythagorean-triples 3) ; (9 12 15)
  (stream-ref pythagorean-triples 4) ; (8 15 17))
```

Note even with the above optimization of computing (pairs t u) only once, finding pythagorean triples is very slow: (8 15 17) is the 163813th item in the triples-integers stream.

Note also that it may be possible to skip computing first-of-integer-pair entirely, by inverting the result from exercise 3.66:  $z = (n-m+1/2) 2^m - 2$ , solving for m in terms of z; this would reduce complexity by a factor of 2

Sphinxsky

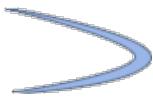
```
(define (triples s t u)
  (define p (pairs s t))
  (define (iter p now-u)
    (let ((car-now-u (stream-car now-u))
          (car-p (stream-car p)))
      (if (<= car-now-u (car car-p))
          (cons-stream
            (cons car-now-u car-p)
            (iter p (stream-cdr now-u))))
          (iter (stream-cdr p) u))))
  (iter p u))

(define pythagorean-triples
  (stream-filter
    (lambda (triple)
      (let ((t (map square triple)))
        (= (+ (car t) (cadr t)) (caddr t))))
    (triples integers integers integers)))
```

In this way, you can only count the number calculated by xdavidliu, and then the memory overflows



# sicp-ex-3.70



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.69) | Index | Next exercise (3.71) >>

x3v

Hardest part is thinking of what to name the procedures..

```
; ; Helper function
(define (stream->list stream n) ; ; n is number of elements to add to list
  (if (= n 0)
      '()
      (cons (stream-car stream) (stream->list (stream-cdr stream) (- n 1)))))

; ; Exercise 3.70
; ; Part A
(define (sum-weight p)
  (+ (car p) (cadr p)))

(define (merge-weighted s1 s2 proc)
  (cond ((stream-null? s1) s2)
        ((stream-null? s2) s1)
        (else
         (let ((s1car (stream-car s1))
               (s2car (stream-car s2)))
           (let ((w1 (proc s1car))
                 (w2 (proc s2car)))
             (if (< w1 w2)
                 (cons-stream s1car (merge-weighted (stream-cdr s1) s2 proc))
                 (cons-stream s2car (merge-weighted s1 (stream-cdr s2) proc)))))))

(define (weighted-pairs s1 s2 proc)
  (cons-stream
    (list (stream-car s1) (stream-car s2))
    (merge-weighted
      (stream-map (lambda (x) (list (stream-car s1) x)) (stream-cdr s2))
      (weighted-pairs (stream-cdr s1) (stream-cdr s2) proc)
      proc)))

; ; test
(define ordered-pairs (weighted-pairs integers integers sum-weight))
(map sum-weight (stream->list ordered-pairs 50))
;; (2 3 4 4 5 5 6 6 6 7 7 7 8 8 8 8 9 ...)

; ; Part B
(define (weight p)
  (+ (* 2 (car p))
     (* 3 (cadr p))
     (* 5 (car p) (cadr p)))))

(define (not-divisible? dividend divisor)
  (not (= 0 (remainder dividend divisor))))

(define (compound-not-divisible? dividend x y z)
  (and (not-divisible? dividend x)
       (not-divisible? dividend y)
       (not-divisible? dividend z)))

(define filtered-integers
  (stream-filter (lambda (x) (compound-not-divisible? x 2 3 5)) integers))

; ; test
(define ordered-conditional-pairs
  (weighted-pairs filtered-integers filtered-integers weight))

(map weight (stream->list ordered-conditional-pairs 50))
;; (10 58 90 106 138 154 186 234 250 280 298 330 346...)
```

xdaidiliu

For merge-weighted, it is important to include \*both\* stream heads if there is ever a tie in weight; we don't simply discard duplicates the way we did for merge.

```
(define (merge-weighted s1 s2 weight)
```

```

(cond ((stream-null? s1) s2)
      ((stream-null? s2) s1)
      (else
        (let ((s1car (stream-car s1))
              (s2car (stream-car s2)))
          (let ((w1 (weight s1car))
                (w2 (weight s2car)))
            (cond ((< w1 w2)
                  (cons-stream s1car
                               (merge-weighted (stream-cdr s1) s2 weight)))
                  ((> w1 w2)
                  (cons-stream s2car
                               (merge-weighted s1 (stream-cdr s2) weight)))
                  (else
                    (cons-stream
                      s1car
                      (cons-stream
                        s2car ; must include both in case of ties!
                        (merge-weighted
                          (stream-cdr s1)
                          (stream-cdr s2)
                          weight))))))))

```

For weighted-pairs, order \*within\* the pair doesn't matter, so we need to stream-map \*twice\*, similarly to how we implemented all-pairs in a previous exercise. (I assume we are supposed to keep it general like this rather than assume that  $i \leq j$ , since the condition " $i \leq j$ " is given in the examples part and part b).

```

;; note this fails if the stream-maps don't respect the weight order
(define (weighted-pairs s t weight)
  (cons-stream
    (list (stream-car s) (stream-car t))
    (merge-weighted
      (merge-weighted
        (stream-map (lambda (x) (list x (stream-car t)))
                    (stream-cdr s))
        (stream-map (lambda (x) (list (stream-car s) x))
                    (stream-cdr t)))
      weight)
    (weighted-pairs (stream-cdr s) (stream-cdr t) weight)
    weight)))

;; examples:

;; part a of exercise

(define pascal-triangle
  (weighted-pairs
    integers
    integers
    (lambda (pair) (apply + pair)))))

(define (partial-stream->list stream n)
  (define (rec str i)
    (if (= i n)
        ()
        (cons (stream-car str)
              (rec (stream-cdr str) (1+ i)))))

  (rec stream 0))

;; utility function to return list of first n items

(partial-stream->list pascal-triangle 16)
;; ((1 1) (2 1) (1 2) (3 1) (2 2) (1 3) (4 1) (3 2) (1 4) (2 3) (5 1) (4 2) (1 5) (3 3) (2 4) (6 1))

;; filter to enforce i <= j

(partial-stream->list
  (stream-filter (lambda (pair) (apply <= pair)) pascal-triangle)
  16)
;; ((1 1) (1 2) (2 2) (1 3) (1 4) (2 3) (1 5) (3 3) (2 4) (1 6) (2 5) (3 4) (1 7) (2 6) (4 4) (3 5))

;; part b

(define (weight-235 pair)
  (let ((i (first pair)) (j (second pair)))
    (+ (* 2 i)
       (* 3 j)
       (* 5 i j)))))

(define all-integer-pairs-by-weight-235
  (weighted-pairs integers integers weight-235))

(map (lambda (pair)
        (cons (weight-235 pair) pair))
  (partial-stream->list

```

```

        all-integer-pairs-by-weight-235
        16))
;; ((10 1 1) (17 2 1) (18 1 2) (24 3 1) (26 1 3) (30 2 2) (31 4 1) (34 1 4) (38 5 1) (42 1
5) (42 3 2) (43 2 3) (45 6 1) (50 1 6) (52 7 1) (54 4 2))

;; now filter in order to satisfy part b condition

(define (not-divisible-by-235? n)
  (not (or (even? n)
            (zero? (remainder n 3))
            (zero? (remainder n 5)))))

(define (part-b-condition? pair)
  (let ((i (first pair)) (j (second pair)))
    (and (<= i j)
         (not-divisible-by-235? i)
         (not-divisible-by-235? j)))))

;; part b result

(partial-stream->list
  (stream-filter
    part-b-condition?
    all-integer-pairs-by-weight-235)
  20)
;; ((1 1) (1 7) (1 11) (1 13) (1 17) (1 19) (1 23) (1 29) (1 31) (7 7) (1 37) (1 41) (1 43)
(1 47) (1 49) (1 53) (7 11) (1 59) (1 61) (7 13)

```

meteorgan

```

(define (merge-weighted s1 s2 weight)
  (cond ((stream-null? s1) s2)
        ((stream-null? s2) s1)
        (else
          (let ((cars1 (stream-car s1))
                (cars2 (stream-car s2)))
            (cond ((< (weight cars1) (weight cars2))
                  (cons-stream cars1
                               (merge-weighted (stream-cdr s1) s2
                                              weight)))
                  ((= (weight cars1) (weight cars2))
                   (cons-stream cars1
                               (merge-weighted (stream-cdr
                                                s1) s2 weight)))
                  (else (cons-stream cars2
                                     (merge-weighted s1 (stream-cdr s2)
                                                    weight)))))))
  (define (weighted-pairs s1 s2 weight)
    (cons-stream (list (stream-car s1) (stream-car s2))
                (merge-weighted (stream-map (lambda (x) (list (stream-car s1) x))
                                            (stream-cdr s2))
                               (weighted-pairs (stream-cdr s1) (stream-cdr s2)
                                              weight)))))

  (define weight1 (lambda (x) (+ (car x) (cadr x))))
  (define pairs1 (weighted-pairs integers integers weight1))

  (define weight2 (lambda (x) (+ (* 2 (car x)) (* 3 (cadr x)) (* 5 (car x) (cadr x)))))
  (define (divide? x y) (= (remainder y x) 0))
  (define stream235
    (stream-filter (lambda (x) (not (or (divide? 2 x) (divide? 3 x) (divide? 5 x))))
                  integers))
  (define pairs2 (weighted-pairs stream235 stream235 weight2)))

```

sam

Another way for part (b) is: use stream from ex3.56 to create weighted-pairs.

beantowel

i think the WEIGHT function takes two arguments rather than one. So i don't understand meteorgan's solution

hoonji

The weight procedure should accept one argument, which is a pair.  
Note that in this exercise, the merge-weighted procedure operates on a stream of pairs.

mart256

```
(define (merge-weighted s1 s2 weight)
  (cond ((stream-null? s1) s2)
        ((stream-null? s2) s1)
        (else
          (let ((s1car (stream-car s1)) (s2car (stream-car s2)))
            (cond ((< (weight s1car) (weight s2car))
                  (cons-stream
                    s1car
                    (merge-weighted (stream-cdr s1) s2 weight)))
                  ((> (weight s1car) (weight s2car))
                  (cons-stream
                    s2car
                    (merge-weighted s1 (stream-cdr s2) weight)))
                  (else
                    (cons-stream
                      s1car
                      (cons-stream
                        s2car
                        (merge-weighted (stream-cdr s1)
                                      (stream-cdr s2) weight))))))))))
```

Sphinxsky

```
(define (merge-weighted stream1 stream2 weight)
  (cond ((stream-null? stream1) stream2)
        ((stream-null? stream2) stream1)
        (else
          (let ((stream1-car (stream-car stream1))
                (stream2-car (stream-car stream2)))
            (if (> (weight stream1-car) (weight stream2-car))
                (cons-stream
                  stream2-car
                  (merge-weighted stream1 (stream-cdr stream2) weight))
                (cons-stream
                  stream1-car
                  (merge-weighted (stream-cdr stream1) stream2 weight)))))))

(define (weighted-pairs s t weight)
  (cons-stream
    (list (stream-car s) (stream-car t))
    (merge-weighted
      (stream-map (lambda (x) (list (stream-car s) x)) (stream-cdr t))
      (weighted-pairs (stream-cdr s) (stream-cdr t) weight)
      weight)))

; a)
(weighted-pairs integers integers (lambda (pair) (apply + pair)))

; b)
(define 2-3-5-number
  (merge
    (merge
      (scale-stream integers 2)
      (scale-stream integers 3))
    (scale-stream integers 5)))

(weighted-pairs
  2-3-5-number
  2-3-5-number
  (lambda (pair)
    (let ((i (car pair))
          (j (cadr pair)))
      (+ (* 2 i) (* 3 j) (* 5 i j)))))
```

Thomas

@Sphinxsky answer for b is incorrect. there i and j are only divisible by by 2,3 or 5, but they should both not be divisible by either of these numbers. Meteorgans solution is correct (didn't bother to check the rest)

dekraai

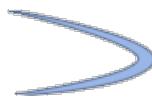
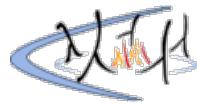
Isn't all answers incorrect, considering that the first pair is never compared?

krubar

First pair is correct by definition, see the footnote 197:

We will require that the weighting function be such that the weight of a pair increases as we move out along a row or down along a column of the array of pairs.

# sicp-ex-3.71



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.70) | Index | Next exercise (3.72) >>

meteorgan

```
(define (Ramanujan s)
  (define (stream-cadr s) (stream-car (stream-cdr s)))
  (define (stream-cddr s) (stream-cdr (stream-cdr s)))
  (let ((scar (stream-car s))
        (scadr (stream-cadr s)))
    (if (= (sum-triple scar) (sum-triple scadr))
        (cons-stream (list (sum-triple scar) scar scadr)
                     (Ramanujan (stream-cddr s))))
        (Ramanujan (stream-cdr s))))
  (define (triple x) (* x x x))
  (define (sum-triple x) (+ (triple (car x)) (triple (cadr x))))
  (define Ramanujan-numbers
    (Ramanujan (weighted-pairs integers integers sum-triple)))

the next five numbers are:
(4104 (2 16) (9 15))
(13832 (2 24) (18 20))
(20683 (10 27) (19 24))
(32832 (4 32) (18 30))
(39312 (2 34) (15 33))
```

x3v

```
(define (sum-of-cubes p)
  (+ (cube (car p)) (cube (cadr p)))))

(define ordered-sum-of-cubes
  (stream-map sum-of-cubes (weighted-pairs integers integers sum-of-cubes)))

(define (ramanujan-stream s)
  (let ((next (stream-cdr s)))
    (if (= (stream-car s) (stream-car next))
        (cons-stream (stream-car s)
                     (ramanujan-stream next))
        (ramanujan-stream next)))

(stream->list (ramanujan-stream ordered-sum-of-cubes) 6)
;; (1729 4104 13832 20683 32832 39312)
```

land

```
(define ram-stream
  (stream-map cdr
              (stream-filter (lambda (x) (= (cube-weight (car x)) (cadr x)))
                            (stream-map list cube-stream (stream-cdr (stream-map cube-weight cube-
stream))))))

;Wanted to do the exercise completely using stream paradigm.
```

hi-artem

"Easy to understand" iterative version :-)

```
(define (cubicSum p)
  (let ((i (car p))
        (j (cadr p)))
```

```

(+ (* i i i) (* j j j)))

(define sortedStream (weighted-pairs cubicSum (integers 0) (integers 0)))

(define (ramanujan s pre)
  (let ((num (cubicSum (stream-car s))))
    (cond ((= pre num)
           (cons-stream
            num
            (ramanujan (stream-cdr s) num)))
          (else
           (ramanujan (stream-cdr s) num)))))

(display-stream (ramanujan sortedStream 0))

```

karthikk

Small bug in meteorgan's solution: You don't want to apriori exclude the possibility of three consequent pairs all having the same weight...i.e. you dont want to recurse into the cddr in case you find a pair with equal weights, because (weight cdr) might be equal to (weight caddr)...

Shawn

But this would give you the same number as the one you find with car and cadr. So we are safe to skip to cddr directly.

Zeyang

My solution...

The function "repeated" should be defined in chapter one I think..

```

(define (get-special-number-stream different-ways)
  (define (proc p1 p2 p3)
    (define (sum-of-cubed x y) (+ (cube x) (cube y)))
    (let ((p1-sum (sum-of-cubed (car p1) (cadr p1)))
          (p2-sum (sum-of-cubed (car p2) (cadr p2)))
          (p3-sum (sum-of-cubed (car p3) (cadr p3))))
      (if (and (= p2-sum p1-sum)
               (not (= p2-sum p3-sum)))
          p2-sum
          false)))
    (stream-filter (lambda (x) x)
      (stream-map proc
        ordered-pairs
        ((repeated stream-cdr (- different-ways 1)) ordered-pairs)
        ((repeated stream-cdr different-ways) ordered-pairs)))

  (define ramanujan-numbers
    (get-special-number-stream 2)))

```

Sphinxsky

```

(define (ramanujan-numbers)
  (define (cubic-sum pair) (+ (expt (car pair) 3) (expt (cadr pair) 3)))
  (define cubic-pairs (weighted-pairs integers integers cubic-sum))
  (define (rec s1 s2)
    (let ((car-s1 (stream-car s1))
          (car-s2 (stream-car s2)))
      (if (= (cubic-sum car-s1) (cubic-sum car-s2))
          (cons-stream
           car-s1
           (rec (stream-cdr s1) (stream-cdr s2)))
          (rec (stream-cdr s1) (stream-cdr s2)))))

  (stream-map cubic-sum (rec (stream-cdr cubic-pairs) cubic-pairs)))

(show-streams 6 (ramanujan-numbers))

; (1729)
; (4104)
; (13832)

```

```
; (20683)
; (32832)
; (39312)
```

master

How about this?

```
(define (sum-of-cubes x y)
  (+ (cube x) (cube y)))

(define (ramanujan? x y)
  (= (sum-of-cubes (car x) (cadr x))
     (sum-of-cubes (car y) (cadr y)))))

(define ramanujan (stream-filter (lambda (x y) (ramanujan? x y)) (weighted-pairs integers
  integers sum-of-cubes)))
```

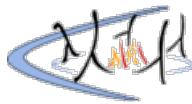
krubar

I don't see it could work out, since stream-filter takes single-argument function. Your stream-filter implementation would need to take and compare two consecutive pairs from weighter-pairs stream.

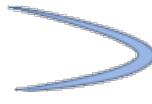
>>>

---

Last modified : 2021-11-19 13:39:53  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-3.72



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.71) | Index | Next exercise (3.73) >>

x3v

Constructs a new stream with output in the form of (sum-of-squares, pair1, pair2, pair3)

```
;; Exercise 3.72
(define (sum-of-squares p)
  (+ (square (car p)) (square (cadr p)))))

(define ordered-pairs
  (weighted-pairs integers integers sum-of-squares))

(define (equiv-sum-squares-stream s)
  (let ((next-1 (stream-cdr s))
        (next-2 (stream-cdr (stream-cdr s))))
    (let ((p1 (stream-car s))
          (p2 (stream-car next-1))
          (p3 (stream-car next-2)))
      (let ((x1 (sum-of-squares p1))
            (x2 (sum-of-squares p2))
            (x3 (sum-of-squares p3)))
        (if (= x1 x2 x3)
            (cons-stream
              (list x1 p1 p2 p3)
              (equiv-sum-squares-stream (stream-cdr next-2)))
            (equiv-sum-squares-stream next-1))))))

(stream->list (equiv-sum-squares-stream ordered-pairs) 5)
;; ((325 (10 15) (6 17) (1 18))
;; (425 (13 16) (8 19) (5 20))
;; (650 (17 19) (11 23) (5 25))
;; (725 (14 23) (10 25) (7 26))
;; (845 (19 22) (13 26) (2 29)))
```

dzy

```
(define (weight s)
  (let ((i (car s))
        (j (cadr s)))
    (+ (expt i 2) (expt j 2))))
(define (merge-weighted s t weight)
  (cond ((stream-null? s) t)
        ((stream-null? t) s)
        (else
          (let ((s0 (stream-car s))
                (t0 (stream-car t)))
            (cond ((< (weight s0) (weight
t0))
                    (cons-stream s0
                      (merge-weighted (stream-cdr s) t weight)))
                  (else
                    (cons-stream t0
                      (merge-weighted (stream-cdr t) s weight)))))))
(define (pairs-weighted s t weight)
  (cons-stream
    (list (stream-car s) (stream-car t))
    (merge-weighted
      (stream-map (lambda (x) (list (stream-car s) x))
                  (stream-cdr t))
      (pairs-weighted (stream-cdr s) (stream-cdr t) weight)
      weight)))
(define (get-number s)
  (let ((last-number (stream-ref s 0))
        (s0 (stream-ref s 1))
        (s1 (stream-ref s 2))
        (s2 (stream-ref s 3)))
    ...))
```

```

(s3 (stream-cdr (stream-cdr s))))
(cond ((or (= last-number (weight s0))
           (not (= (weight
s0) (weight s1) (weight s2)))))
       (get-number (cons-stream last-
number s3)))
      (else (cons-stream (list (weight s0) s0
s1 s2)

(get-number (cons-stream (weight s0) s3))))))

(define s (pairs-weighted integers integers weight))
(define q (get-number (cons-stream 0 s)))
(display-stream q 10)
;(325 (1 18) (6 17) (10 15))
;(425 (5 20) (8 19) (13 16))
;(650 (5 25) (11 23) (17 19))
;(725 (7 26) (10 25) (14 23))
;(845 (2 29) (13 26) (19 22))
;(850 (3 29) (11 27) (15 25))
;(925 (5 30) (14 27) (21 22))
;(1025 (1 32) (8 31) (20 25))
;(1105 (4 33) (9 32) (12 31))
;(1250 (5 35) (17 31) (25 25))

```

### zerocooldown

```

(define (weighted-pairs s t weight)
  (cons-stream
    (list (stream-car s) (stream-car t))
    (merge-weighted
      (stream-map (lambda (x)
                    (list (stream-car s) x))
                  (stream-cdr t))
      (weighted-pairs (stream-cdr s) (stream-cdr t) weight)
      weight)))

(define (collect-consecutive-pairs-n s n evaluator-n)
  (let ((next-n (take-n s n)))
    (if (evaluator-n next-n)
        (cons-stream next-n
                    (collect-consecutive-pairs-n (stream-cdr s)
                                                n
                                                evaluator-n))
        (collect-consecutive-pairs-n (stream-cdr s)
                                    n
                                    evaluator-n)))

(define (square-sum p)
  (let ((i (car p))
        (j (cadr p)))
    (+ (* i i)
       (* j j)))))

(define 3-way-square-sum
  (stream-map (lambda (t)
                (list (square-sum (car t)) t))
              (collect-consecutive-pairs-n (weighted-pairs integers
                                                integers
                                                square-sum)
                                          3
                                          (lambda (t)
                                            (= (square-sum (car t))
                                               (square-sum (cadr t))
                                               (square-sum (caddr t)))))))

```

### rmn

```

(define (square-weight x y)
  (+ (* x x) (* y y)))

(define squared-ints-pairs
  (weighted-pairs integers integers (lambda (x) (apply square-weight x)))))

(define (take-stream stream n)
  (if (= n 0)

```

```

'()
(cons (stream-car stream)
      (take-stream (stream-cdr stream) (- n 1)))))

(define (two-squares-three-ways-helper s weight)
  (let ((vals (take-stream s 3)))
    (if (apply = (map (lambda (x) (apply weight x)) vals))
        (cons-stream (apply weight (car vals))
                     (two-squares-three-ways-helper (stream-cdr s) weight))
        (two-squares-three-ways-helper (stream-cdr s) weight)))

(define two-squares-three-ways
  (two-squares-three-ways-helper squared-ints-pairs square-weight))

```

```

(take-stream two-squares-three-ways 5)
; (325 425 650 725 845)

```

seek

A generic procedure for problems like 3.71 and 3.72.

```

;; s : A stream of lists sorted in ascending order w.r.t aggr-fn.
;; aggr-fn : A function used to sort lists of s. Receives one list as a parameter.
;;           Return value doesn't have to be an interger as long as it has an ordering.
;; n : Minimum number of consecutive elts.
(define (consecutive-elts-from-stream s aggr-fn n)
  (define (skip-same-elts s val)
    (if (equal? (aggr-fn val) (aggr-fn (stream-car s)))
        (skip-same-elts (stream-cdr s) val)
        s))

  (define (build-same-elts-list from to)
    (let ((first (stream-car from))
          (next (stream-cdr from)))
      (cond ((null? to)
              (build-same-elts-list next (list first)))
            ((equal? (aggr-fn first) (aggr-fn (car to)))
             (build-same-elts-list next (cons first to)))
            (else to)))

    (if (equal? (aggr-fn (stream-car s))
                (aggr-fn (stream-ref s (- n 1))))
        (cons-stream (cons (aggr-fn (stream-car s))
                           (build-same-elts-list s ()))
                      (consecutive-elts-from-stream (skip-same-elts s (stream-car s))
                        aggr-fn n))
        (consecutive-elts-from-stream (skip-same-elts s (stream-car s)) aggr-fn n)))

;; 3.72
(define (sum-squares li) (apply + (map square li)))
(define sum-square-ordered-pairs
  (weighted-pairs sum-squares integers))

(define pseudo-ramanujan
  (consecutive-elts-from-stream sum-square-ordered-pairs sum-squares 3))

(display-stream pseudo-ramanujan)
; (325 (1 18) (6 17) (10 15))
; (425 (5 20) (8 19) (13 16))
; (650 (5 25) (11 23) (17 19))
; (725 (7 26) (10 25) (14 23))
; (845 (2 29) (13 26) (19 22))
; (850 (3 29) (11 27) (15 25))
; (925 (5 30) (14 27) (21 22))
; (1025 (1 32) (8 31) (20 25))
; (1105 (4 33) (9 32) (12 31) (23 24))
; (1250 (5 35) (17 31) (25 25))
; ...

;; 3.71
(define (cube x) (* x x x))
(define (sum-cubes li) (apply + (map cube li)))

(define sum-cube-ordered-pairs
  (weighted-pairs sum-cubes integers))

(define ramanujan
  (consecutive-elts-from-stream sum-cube-ordered-pairs sum-cubes 2))

(display-stream ramanujan)
; (1729 (1 12) (9 10))
; (4104 (2 16) (9 15))
; (13832 (2 24) (18 20))

```

```
; (20683 (10 27) (19 24))
; (32832 (4 32) (18 30))
; (39312 (2 34) (15 33))
; (40033 (9 34) (16 33))
; (46683 (3 36) (27 30))
; (64232 (17 39) (26 36))
; (65728 (12 40) (31 33))
; ...
```

Sphinxsky

```
(define (triple-ways-square-sum-numbers)
  (define (square-sum pair) (apply + (map square pair)))
  (define square-pairs (weighted-pairs integers integers square-sum))
  (define (rec . streams)
    (let ((car-streams (map stream-car streams)))
      (if (apply = (map square-sum car-streams))
          (cons-stream
            car-streams
            (apply rec (map stream-cdr streams)))
          (apply rec (map stream-cdr streams)))))

  (stream-map
    (lambda (pairs)
      (cons (square-sum (car pairs)) pairs))
    (rec
      square-pairs
      (stream-cdr square-pairs)
      (stream-cdr (stream-cdr square-pairs)))))
```

master

Again, is this not sufficient?

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

(define (sum-squares-three-ways? x y z)
  (= (sum-of-squares (car x) (cadr x))
     (sum-of-squares (car y) (cadr y))
     (sum-of-squares (car z) (cadr z)))))

(define sum-squares-three-ways (stream-filter (lambda (x y z) (sum-squares-three-ways? x
y z))
                                             (weighted-pairs integers
integers
sum-of-squares)))
```

# sicp-ex-3.73

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.72) | Index | Next exercise (3.74) >>

meteorgan

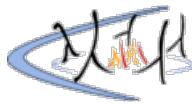
```
(define (RC r c dt)
  (define (proc i v)
    (add-streams (scale-stream i r)
                 (integral (scale-stream i (/ 1 c)) v dt)))
  proc)
```

mathieuborderé @meteorgan I think it's better to just return a lambda, naming the function in the example before the actual exercise was necessary to recursively call it, here it's not needed.

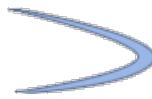
Beckett

```
(define (RC r c dt)
  (lambda (si initial-voltage)
    (add-stream (scale-stream si R)
               (integral (scale-stream si (/ 1 C)) initial-voltage dt))))
```

Last modified : 2022-02-23 02:27:23  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-3.74



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.73) | Index | Next exercise (3.75) >>

meteorgan

```
(define zero-crossings
  (stream-map sign-change-detector sense-data (cons-stream 0 sense-data)))
```

karthikk

I don't think that is right. If the first element of the sense-data stream were negative above solution will show a sign-change crossing immediately when there is none.

```
(define (zero-crossings sense-data)
  (stream-map sign-change-detector sense-data (stream-cdr sense-data)))
```

hoonji

meteorgan's solution complies with the original zero-crossings in the book. The first two arguments to sign-change-detector should be: (stream-car sense-data) and 0

xdaavidliu

I agree with hoonji that meteorgan's solution correctly answers the problem. However I would like to point out that the last-value parameter is completely superfluous when the next value in the stream is always easily accessible. In this manner, we can rewrite make-zero-crossings elegantly to require only a single argument:

```
(define (make-zero-crossings input-stream)
  (stream-map sign-change-detector (stream-cdr sense-data) sense-data))
```

karthikk's solution addresses this issue too, but as pointed out already, that is not what the book is asking for literally in this exercise. In my opinion this seems like a rare example of lapse of judgment on the part of the authors.

Sphinxsky

```
(define zero-crossings
  (stream-map
    sign-change-detector
    sense-data
    (cons-stream
      (stream-car sense-data)
      sense-data)))
```

Cirno

```
;; helpers
(define (sign-change-detector old new)
  (cond ((or (positive? old) (zero? old))
         (if (negative? new) -1 0))
        ((negative? old)
         (if (negative? new) 0 1)))

(define (random-in-range low high)
  (let ((range (- high low)))
    (+ low (* (random) range)))

(define (random-stream low high)
  (cons-stream (random-in-range low high)
               (random-stream low high)))
```

```

(define sense-data (random-stream -10 10))

(define (make-zero-crossings input-stream last-value)
  (cons-stream
    (sign-change-detector
      (stream-car input-stream)
      last-value)
    (make-zero-crossings
      (stream-cdr input-stream)
      (stream-car input-stream)))))

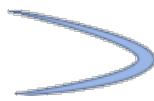
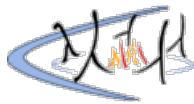
(define zero-crossings-alysa
  (make-zero-crossings sense-data 0))

(define zero-crossings-eva
  (stream-map sign-change-detector
    sense-data
    (stream-cdr sense-data)))

(display "eva:")
(newline)
(stream-ref zero-crossings-eva 0)
(stream-ref zero-crossings-eva 1)
(stream-ref zero-crossings-eva 2)
(stream-ref zero-crossings-eva 3)
(stream-ref zero-crossings-eva 4)
(stream-ref zero-crossings-eva 5)
(stream-ref zero-crossings-eva 6)
(stream-ref zero-crossings-eva 7)
(stream-ref zero-crossings-eva 8)
(stream-ref zero-crossings-eva 9)
(stream-ref zero-crossings-eva 10)

(display "alyssa:")
(newline)
(stream-ref zero-crossings-eva 0)
(stream-ref zero-crossings-eva 1)
(stream-ref zero-crossings-eva 2)
(stream-ref zero-crossings-eva 3)
(stream-ref zero-crossings-eva 4)
(stream-ref zero-crossings-eva 5)
(stream-ref zero-crossings-eva 6)
(stream-ref zero-crossings-eva 7)
(stream-ref zero-crossings-eva 8)
(stream-ref zero-crossings-eva 9)
(stream-ref zero-crossings-eva 10)

```



<< Previous exercise (3.74) | Index | Next exercise (3.76) >>

meteorgan

```
(define (make-zero-crossings input-stream last-value last-avpt)
  (let ((avpt (/ (+ (stream-car input-stream) last-value) 2)))
    (cons-stream (sign-change-detector avpt last-avpt)
      (make-zero-crossings (stream-cdr input-stream)
        (stream-car input-stream)
        avpt))))
```

xdaividliu

The above solution is correct. However, in my opinion the authors of SICP have made some superfluous and unnecessary choices for this exercise as well as this entire section on zero crossings: keeping track of the last value is completely unnecessary because you have the \*next\* values.

Hence, a smoothed make-zero-crossings requiring only the input-stream as an argument can be implemented like this:

```
(define (stream-cadr s)
  (stream-car (stream-cdr s)))
(define (stream-caddr s)
  (stream-cadr (stream-cdr s)))
(define (average x y) (/ (+ x y) 2))

(define (make-zero-crossings s)
  (let ((val1 (stream-car s))
        (val2 (stream-cadr s))
        (val3 (stream-caddr s)))
    (let ((avg1 (average val1 val2))
          (avg2 (average val2 val3)))
      (cons-stream
        (sign-change-detector avg2 avg1)
        (make-zero-crossings (stream-cdr s))))))
```

Or alternatively, using stream-map as in the previous exercise:

```
(define (average-sign-change-detector x y z)
  (let ((avg1 (average x y))
        (avg2 (average y z)))
    (sign-change-detector avg2 avg1)))

(define (stream-cddr s)
  (stream-cdr (stream-cdr s)))
(define (make-zero-crossings s)
  (stream-map average-sign-change-detector s (stream-cdr s) (stream-cddr s)))
```

yd

Since

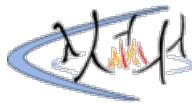
```
(let ((x x-val)) ((lambda (. args) <expr>) args-val))
```

is nothing else but the syntax sugar of

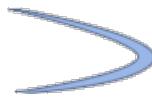
```
((lambda (x . args) <expr>) x-val args-val)
```

,there are only style differences between whether passing "last value" explicitly.





# sicp-ex-3.76



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.75) | Index | Next exercise (3.77) >>

meteorgan

```
(define (smooth s)
  (stream-map (lambda (x1 x2) (/ (+ x1 x2) 2))
              (cons-stream 0 s)
              s))
(define (make-zero-crossings input-stream smooth)
  (let ((after-smooth (smooth input-stream)))
    (stream-map sign-change-detector after-smooth (cons-stream 0 after-smooth))))
```

karthikk

Again no need to cons-stream 0 in this case.

This had to be done in the non-modular (i.e. without using stream-map) 3.74 and 3.75 cases because the function had to be provided with an (arbitrary) initial value of 0 (and presumably the first couple of elements of output stream discarded)

With stream-map, there is no need for an arbitrary initial value for the sense-data: Thus:

```
(define (smooth input-stream)
  (stream-map (lambda(x y) (/ (+ x y) 2)) input-stream (stream-cdr input-stream)))

(define (zero-crossings input-stream)
  (stream-map sign-change-detector input-stream (stream-cdr input-stream)))

(define (smoothed-zero-crossing sense-data)
  (zero-crossings (smooth sense-data)))
```

Sphinxsky

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

(define (smooth stream proc) (stream-map proc (stream-cdr stream) stream))

(define (sign-change-detector now before)
  (cond ((and (> before 0) (< now 0)) (- 1))
        ((and (< before 0) (> now 0)) 1)
        (else 0)))

(define (make-zero-crossings input-stream)
  (stream-map
    sign-change-detector
    (stream-cdr input-stream)
    input-stream))

(define zero-crossings (make-zero-crossings (smooth sense-data average))))
```

joe w

I disagree that we should leave off the initial zero, without it you lose the zero crossing of the first two proper elements of the stream.

```
(define sense-data (cons-stream 5 (cons-stream -8 (cons-stream -7 (cons-stream -1 sense-data)))))

(define (smooth input-stream)
  (define (avg x y)
    (/ (+ x y) 2))
```

```
(s-map avg input-stream (tail input-stream))

(display-stream-until 6 (smooth sense-data))

(define (make-zero-crossings-modular input-stream)
  (let ((smoothed (cons-stream (head input-stream) (smooth input-stream))))
    (s-map sign-change-detector (tail smoothed) smoothed)))

(define zero-crossings-modular
  (make-zero-crossings-modular sense-data))

(display-stream-until 6 zero-crossings-modular)
```

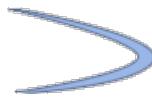
Using the sense data above we get: -1 0 0 1 -1 0 0

If you drop the cons-stream you don't get the zero crossing for 5 to -8; the result stream starts at the first 0 for the sensing data provided.

This assumes the stream starts in an off state, so we start with a zero, if it were an on stream that we just picked up at our first sensing data we could cons the head of the sensing data onto itself.



# sicp-ex-3.77



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.76) | Index | Next exercise (3.78) >>

meteorgan

```
(define (integral delayed-integrand initial-value dt)
  (cons-stream initial-value
    (let ((integrand (force delayed-integrand)))
      (if (stream-null? integrand)
          the-empty-stream
          (integral (delay (stream-cdr integrand))
                    (+ (* dt (stream-car integrand))
                       initial-value)
                    dt)))))
```

Last modified : 2012-05-22 07:17:25  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-3.78

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.77) | Index | Next exercise (3.79) >>

meteorgan

```
(define (solve-2nd a b dt y0 dy0)
  (define y (integral (delay dy) y0 dt))
  (define dy (integral (delay ddy) dy0 dt))
  (define ddy (add-streams (scale-stream dy a) (scale-stream y b)))
  y)
```

seok

```
; Some test cases to verify an implementation..

(stream-ref (solve-2nd 1 0 0.0001 1 1) 10000) ; e
(stream-ref (solve-2nd 0 -1 0.0001 1 0) 10472) ; cos pi/3 = 0.5
(stream-ref (solve-2nd 0 -1 0.0001 0 1) 5236) ; sin pi/6 = 0.5
```

asmn

I had the same solution has meteorgan and the program runs out of memory when trying to solve anything. I have memory limit to 1GB as well

Last modified : 2020-09-16 12:57:58  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-3.79

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.78) | Index | Next exercise (3.80) >>

meteorgan

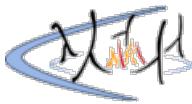
```
(define (general-solve-2nd f y0 dy0 dt)
  (define y (integral (delay dy) y0 dt))
  (define dy (integral (delay ddy) dy0 dt))
  (define ddy (stream-map f dy y))
  y)
```

Eva Lu Ator

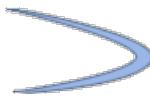
The first argument to f should be dy/dt rather than dy. Therefore we have

```
(define (solve-2nd-general f dt y0 dy0)
  (define y (integral (delay dy) y0 dt))
  (define dy (integral (delay ddy) dy0 dt))
  (define ddy (stream-map f (scale-stream dy (/ 1.0 dt)) y))
  y)
```

Last modified : 2022-09-11 15:49:25  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-3.80



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.79) | Index | Next exercise (3.81) >>

meteorgan

```
(define (RLC R L C dt)
  (define (rcl vc0 il0)
    (define vc (integral (delay dvc) vc0 dt))
    (define il (integral (delay dil) il0 dt))
    (define dvc (scale-stream il (- (/ 1 C))))
    (define dil (add-streams (scale-stream vc (/ 1 L))
                             (scale-stream il (- (/ R
L))))))
    (define (merge-stream s1 s2)
      (cons-stream (cons (stream-car s1) (stream-car s2))
                   (merge-stream (stream-cdr s1) (stream-
cadr s2))))
    (merge-stream vc il)))
  rcl)
```

Rpx

why not simply (cons vc il) at the end of the procedure instead of merging? the exercise says "a pair of the streams".

>>>>>

atupal

We could first compute the integral then scale the stream

```
(define (RLC R L C dt)
  (define (proc vc0 il0)
    (define vc (scale-stream (integral (delay il) (* (- C) vc0) dt) (/ -1 C)))
    (define il (integral (delay dil) il0 dt))
    (define dil (add-streams (scale-stream il (/ (- R) L))
                            (scale-stream vc (/ 1 L)))))
    (stream-map cons vc il)))
  proc)
```

karthikk

Two quick points:

- 1) I interpreted the "pair of streams" as a simple consing instead of a mapping/merging (which produced a "stream of pairs").
- 2) More importantly, the hack mentioned in 4.1.6 seems necessary to run this in the #lang sicp environment in Racket so I have written out my solution in case you find too that your solution doesn't work in Racket...

Now the solution doesn't really need the definitions of dvC and diL so four lines could be removed from below code but I think the longer code is clearer...

```
(define (RLC R L C dt)
  (lambda (vC-init iL-init)
    (let ((vC '*unassigned*)
          (iL '*unassigned*)
          (dvC '*unassigned*)
          (diL '*unassigned*))
      (set! vC (d-integral (delay dvC) vC-init dt))
      (set! iL (d-integral (delay dil) iL-init dt))
      (set! dvC (stream-map (lambda(x) (/ (- x) C)) iL))
      (set! diL (stream-map (lambda (x y) (+ (/ x L) (/ (* (- R) y) L))) vC iL))
      (cons vC iL))))
```

Thank you so much karthikk. I met the same issue in the #lang sicp env in Racket.

Dewey

Shawn

Here is my solution. The exercise asks for pair of streams instead of stream of pairs. Therefore, I just cons-ed the two streams instead of doing mapping on two streams.

```
(define (RLC R L C dt)
  (lambda (vC0 iL0)
    (define vC (integral (delay dVC) vC0 dt))
    (define iL (integral (delay diL) iL0 dt))
    (define diL (add-streams (scale-stream vC (/ 1 L))
                             (scale-stream iL (/ (- R) L))))
    (define dVC (scale-stream iL (/ -1 C)))
    (cons vC iL)))
```

Joe W

While I ultimately ended up with Shawn's solution, I initially came up with this solution which has some cool Time Lord shit in it.

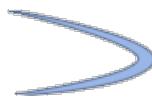
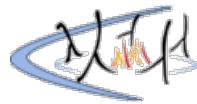
I wanted to delay the argument to scale-streams to create dVC but since scale-streams doesn't have a force I just delayed the entire scaling operation. Then the line before dVC is defined I use it as an argument, but I must delay it. However it contains a delayed operation itself I must wrap that in a force call, which is wrapped in a delay! Pretty cool that you can do this, and while I did get to a simpler solution it's easy to see how combining strict and lazy evaluation can become problematic.

```
(define (RLC R L C dt)
  (define (RLC-proc vC0 iL0)
    (define vC (integral-delay-from (delay (force dVC)) vC0 dt))
    (define dVC (delay (scale-stream iL (/ -1 C))))
    (define iL (integral-delay-from (delay diL) iL0 dt))
    (define diL (add-streams (scale-stream vC (/ 1 L))
                             (scale-stream iL (/ (* -1 R) L))))
    (cons vC iL)))
  RLC-proc)
})}">>>>
```

Eva Lu Ator

An alternative approach would be to derive an equivalent second order differential equation, solve it and differentiate to get the stream for the other variable

# sicp-ex-3.81



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.80) | Index | Next exercise (3.82) >>

meteorgan

```
(define (random-update x)
  (remainder (+ (* 13 x) 5) 24))
(define random-init (random-update (expt 2 32)))

;; assume the operation 'generator' and 'reset' is a stream,
;; and if the command is 'generator', the element of
;; stream is a string, if the command is 'reset',
;; it is a pair whose first element is 'reset',
;; the other element is the reset value.
(define (random-number-generator command-stream)
  (define random-number
    (cons-stream random-init
      (stream-map (lambda (number command)
        (cond ((null? command)
          (the-empty-stream)
          'generator)
          (update number))
        (command)
        (car command) 'reset)))
      command)))
  (command -- "command")))

(random-number)
```

hi-artem

I think this code has a bug:

First element in generated stream is always the same - "random-init", even if command stream started with 'reset OTHER\_NUMBER'

xy

What about this:

```
(define (rand request-stream)
  (let ((req (stream-car request-stream)))
    (let ((random-init (if (eq? 'reset (car req))
      (cadr req)
      random-init)))
      (request-stream (if (eq? 'reset (car req))
        (stream-cdr request-stream)
        request-stream)))
    (define random-numbers
      (cons-stream random-init
        (stream-map
          (lambda (req rnum)
            (cond ((eq? 'generate (car req))
              (rand-update rnum))
              ((eq? 'reset (car req))
                (cadr req))
              (else (error "Wrong request -- RAND" req))))
            request-stream random-numbers)))
      random-numbers)))
```

```

;;; Test
(define s1 (rand (stream '(reset 2010)
                           '(generate)
                           '(generate)
                           '(generate)
                           '(reset 2020)
                           '(generate)
                           '(generate)
                           '(reset 1234)
                           '(generate)
                           '(generate) )))

;Value: s1

(map (lambda (e)
          (stream-ref s1 e))
     (iota 10))
;Value 222: (2010 67 57 41 2020 83 108 1234 70 11)

```

mathieuborderé

```

(define (rand-generator commands)
  (define (rand-helper num remaining-commands)
    (let ((next-command (stream-car remaining-commands)))
      (cond ((eq? next-command 'generate)
             (cons-stream num
                           (rand-helper (rand-update num)
                                       (stream-cdr remaining-commands))))
            ((pair? next-command)
             (if (eq? (car next-command) 'reset)
                 (cons-stream (cdr (stream-car remaining-commands))
                             (rand-helper (rand-update (cdr (stream-car remaining-commands)))
                                         (stream-cdr remaining-commands)))
                 (error "bad command -- " next-command)))
             (else (error "bad command -- " next-command))))))
    (rand-helper rand-init commands))

;;; testing

;;; generate stream of commands
(define gen-stream
  (cons-stream
    (cons 'reset 12)
    (cons-stream 'generate
                  (cons-stream (cons 'reset 100)
                               (cons-stream 'generate
                                             gen-stream)))))

(define rands (rand-generator gen-stream))

(stream-ref rands 0)
(stream-ref rands 1)
(stream-ref rands 2)
(stream-ref rands 3)
(stream-ref rands 4)
(stream-ref rands 5)

;output:

;(stream-ref rands 0)
;Value: 12

;(stream-ref rands 1)
;Value: 1033878523

;(stream-ref rands 2)
;Value: 100

;(stream-ref rands 3)
;Value: 1180356723

;(stream-ref rands 4)
;Value: 12

; (stream-ref rands 5)
;Value: 1033878523

```

karthikk

Here is a different solution from mathiebordere's that separates the dispatch on message from the stream construction:

```

(define (random-numbers continue-val input-stream)
  (define (constructor start-val)
    (cons-stream start-val
      (random-numbers (rand-update start-val)
        (stream-cdr input-stream))))
  (if (stream-null? input-stream)
    'done
    (let ((msg (stream-car input-stream)))
      (cond ((eq? msg 'generate)
             (constructor continue-val))
            ((and (pair? msg) (eq? (car msg) 'reset) (number? (cadr msg)))
             (constructor (cadr msg)))
            (else
             (error "Invalid message found in input-stream" msg)))))))

```

frostov

Here is my solution

```

(define (random-generator requests seed)
  (define s
    (cons-stream
      seed
      (stream-map
        (lambda (request value)
          (cond ((eq? request 'generate) (rand-update value))
                ((and (pair? request) (eq? (car request) 'reset))
                 (cdr request))
                (else (error "random-generator invalid request")))))
      requests
      s)))
  s)

```

awkravchuk

It's also possible to store the reset values along with the commands. Here's (Racket-based) proof-of-concept:

```

(define (random-numbers actions seed)
  (let ((action (stream-car actions)))
    (rest-actions (stream-cdr actions)))
  (cond ((eq? action 'generate)
         (begin
           (random-seed (inexact->exact (floor (* 1000 seed))))
           (let ((random-number (random)))
             (cons-stream
               random-number
               (random-numbers rest-actions random-number)))))

        ((eq? action 'reset)
         (random-numbers
           (stream-cdr rest-actions)
           (stream-car rest-actions)))
        (else (error "Unknown action")))))

(define test-actions
  (cons-stream 'reset (cons-stream 100 (cons-stream 'generate (cons-stream 'generate
  (cons-stream 'reset (cons-stream 100 (cons-stream 'generate (cons-stream 'generate (cons-
  stream 'reset (cons-stream 200 (cons-stream 'generate (cons-stream 'generate
  nil))))))))))))))

(define stream (random-numbers test-actions 0))

(stream-ref stream 0) ; 0.46989804756333914
(stream-ref stream 1) ; 0.2932693988550536
(stream-ref stream 2) ; 0.46989804756333914, reset to the same seed
(stream-ref stream 3) ; 0.2932693988550536
(stream-ref stream 4) ; 0.7791954700538558, reset to different seed
(stream-ref stream 5) ; 0.16903652324331853

```

x davidiu

The purpose of this exercise is to generate a stream of random numbers upon given a stream of \*requests\*, where each element of input request stream is either 'generate or (list request x), where x is some given number.

```

(define (rand-update x)
  (modulo (+ 101 (* x 713)) 53))
  ;; for much better parameters to use here, see

```

```

;; https://en.wikipedia.org/wiki/Linear_congruential_generator#Parameters_in_common_use

(define random-init (rand-update 10))

(define (random-request-generator requests)
  (define (update x request)
    (cond ((eq? request 'generate)
           (rand-update x))
          ((and (pair? request)
                (eq? (car request) 'reset)
                (number? (cadr request)))
           (cadr request))
          (else (error "invalid request" request))))
  (define requested-stream
    (cons-stream
      random-init
      (stream-map update requested-stream requests)))
  requested-stream)

(define example-requests
  (list->stream
    '(generate generate generate (reset 5)
      generate generate (reset 5) generate)))

(stream->list (random-request-generator example-requests))
;; -> (23 17 32 21 5 9 52 5 9)

```

krubar

```

(define (random-numbers-generator requests)
  (define (random-numbers seed)
    (cons-stream seed
      (random-numbers (rand-update seed)))))

(define (generate? request)
  (eq? request 'generate))

(define (reset? request)
  (and (pair? request) (eq? (car request) 'reset)))

(define (loop requests s)
  (cond ((stream-null? requests) the-empty-stream)
        ((generate? (stream-car requests))
         (cons-stream (stream-car s)
           (loop (stream-cdr requests) (stream-cdr s))))
        ((reset? (stream-car requests))
         (let ((r (random-numbers (cadr (stream-car requests)))))
           (cons-stream (stream-car r)
             (loop (stream-cdr requests) (stream-cdr r)))))))

  (loop requests (random-numbers 705894)))

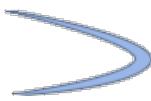
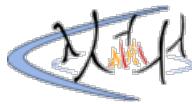
(define requests
  (cons-stream 'generate
    (cons-stream 'generate
      (cons-stream 'generate
        (cons-stream '(reset 705894)
          (cons-stream 'generate
            (cons-stream 'generate
              (the-empty-stream)))))))

(display-stream (random-numbers-generator requests))

;; 705894
;; 1126542223
;; 1579310009
;; 705894
;; 1126542223
;; 1579310009
;; done

```

# sicp-ex-3.82



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (3.81) | Index | Next exercise (4.1) >>

meteorgan

```
(define (random-in-range low high)
  (let ((range (- high low)))
    (+ low (* (random) range))))
(define (random-number-pairs low1 high1 low2 high2)
  (cons-stream (cons (random-in-range low1 high1) (random-in-range low2 high2))
               (random-number-pairs low1 high1 low2 high2)))

(define (monte-carlo experiment-stream passed failed)
  (define (next passed failed)
    (cons-stream (/ passed (+ passed failed))
                (monte-carlo (stream-cdr experiment-stream)
                             passed
                             failed)))
  (if (stream-car experiment-stream)
      (next (+ passed 1) failed)
      (next passed (+ failed 1)))))

(define (estimate-integral p x1 x2 y1 y2)
  (let ((area (* (- x2 x1) (- y2 y1)))
        (randoms (random-number-pairs x1 x2 y1 y2)))
    (scale-stream (monte-carlo (stream-map p randoms) 0 0) area)))

;; test. get the value of pi
(define (sum-of-square x y) (+ (* x x) (* y y)))
(define f (lambda (x) (not (> (sum-of-square (- (car x) 1) (- (cdr x) 1)) 1))))
(define pi-stream (estimate-integral f 0 2 0 2))
```

tango

Meteorgan's solution uses the monte-carlo method with variables 'passed' and 'failed'. Following solution doesn't uses these and relies solely on streams.

```
(define (add-streams s1 s2) (stream-map + s1 s2))
(define ones (cons-stream 1.0 ones))
(define integers (cons-stream 1.0 (add-streams ones integers)))

(define (random-stream lo hi)
  (define (random-in-range low high)
    (let ((range (- high low)))
      (+ low (random range))))
  (cons-stream (random-in-range lo hi) (random-stream lo hi)))

(define (estimate-integral p x1 x2 y1 y2)
  (define throw-results (stream-map (lambda (x) (if (eq? x true) 1.0 0))
                                    (stream-map p (random-stream x1 x2) (random-stream
y1 y2))))
  (define succesful-throws
    (cons-stream (stream-car throw-results) (add-streams (stream-cdr throw-results)
succesful-throws)))
  (define (get-area probability) (* probability (* (- y2 y1) (- x2 x1))))
  (stream-map get-area (stream-map / succesful-throws integers)))
```

master

Isn't producing random numbers the predicate's job? Doesn't the following simple definition of estimate-integral suffice?

```
(define (estimate-integral P x1 x2 y1 y2)
  (let ((width (- x2 x1))
        (height (- y2 y1)))
    (let ((area (* width height)))
      (scale-stream (monte-carlo P 0 0) area))))
```

x3v

Uses the random-in-range procedure previously defined in the book.

```
(define (random-in-range low high)
  (let ((range (- high low)))
    (+ low (random range)))

(define (rand-range-stream low high)
  (cons-stream
    (random-in-range low high)
    (rand-range-stream low high)))

(define (experiment-stream x1 x2 y1 y2 radius)
  (stream-map
    (lambda (x) (> radius x))
    (add-streams
      (stream-map square (rand-range-stream x1 x2))
      (stream-map square (rand-range-stream y1 y2)))))

(define pi-est-stream
  (scale-stream (monte-carlo (experiment-stream -1.0 1.0 -1.0 1.0 1.0) 0 0) 4.0))

(exact->inexact (stream-ref pi-est-stream 50000)) ;; ~3.1429
```

krubar

Similar solution to meteorgan's. Requires random-in-range from Excercise 3.5 and monte-carlo from the beginning of section 3.5.5.

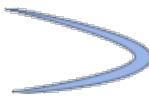
```
(define (randoms-ranged low high)
  (cons-stream (random-in-range low high)
               (randoms-ranged low high)))

(define (integral-estimates P x1 x2 y1 y2)
  (define point-in-integral-stream
    (stream-map P (randoms-ranged x1 x2) (randoms-ranged y1 y2)))
  (monte-carlo point-in-integral-stream 0 0))

(define (in-unit-circle? x y)
  (<= (+ (expt (- x 0.5) 2)
          (expt (- y 0.5) 2))
       (expt 0.5 2)))

(define pi-integral-estimates
  (stream-map (lambda (area) (/ area (* 0.5 0.5)))
             (integral-estimates in-unit-circle? 0.0 1.0 0.0 1.0)))
```

# sicp-ex-4.1



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (3.82) | Index | Next exercise (4.2) >>

meteorgan

```
; left to right
(define (list-of-values1 exps env)
  (if (no-operand? exps)
      '()
      (let* ((left (eval (first-operand exps) env))
             (right (eval (rest-operands exps) env)))
        (cons left right)))

;; right to left
(define (list-of-values2 exps env)
  (if (no-operand? exps)
      '()
      (let* ((right (eval (rest-operands exps) env))
             (left (eval (first-operand exps) env)))
        (cons left right))))
```

craig

99% certain meteorgan's code should have the words "left" and "right" swapped in the let\* in list-of-values2 (the right to left version). Only 99%, so I won't presume to edit it myself.

```
(define (list-of-values-lr exps env)
  (if (no-operands? exps)
      '()
      (let ((first (eval (first-operand exps) env)))
        (let ((rest (list-of-values-lr (rest-operands exps) env)))
          (cons first rest)))))

(define (list-of-values-rl exps env)
  (if (no-operands? exps)
      '()
      (let ((rest (list-of-values-rl (rest-operands exps) env)))
        (let ((first (eval (first-operand exps) env)))
          (cons first rest)))))
```

mathieubordere

```
;;; left-to-right
(define (list-of-values-l2r exps env)
  (if (no-operands? exps)
      '()
      (let ((first-exp (eval (first-operand exps) env)))
        (cons first-exp
              (list-of-values-l2r (rest-operands exps) env)))))

;;; right-to-left
(define (list-of-values-r2l exps env)
  (list-of-values-l2r (reverse exps) env))
```

Sphinxsky

```
; left-to-right
(define (list-of-values exps env)
  (let ((first '()))
    (rest '()))
  (if (no-operands? exps)
```

```

rest
(begin
  (set! first (eval (first-operand exps) env))
  (set! rest (list-of-values (rest-operands exps) env))
  (cons first rest)))))

; right-to-left
(define (list-of-values exps env)
  (let ((first '()))
    (rest '()))
  (if (no-operands? exps)
      rest
      (begin
        (set! rest (list-of-values (rest-operands exps) env))
        (set! first (eval (first-operand exps) env))
        (cons first rest)))))


```

zz I think it's wrong to use cons in right to left.

```

;; ex 4.1
;; left to right
(define (list-of-values-l2r exps env)
  (if (no-operands? exps)
      '()
      (let ((v (eval (first-operand exps) env)))
        (cons v (list-of-values-l2r (cdr exps) env)))))

;; right to left
(define (list-of-values-r2l exps env)
  (reverse (list-of-values-l2r (reverse exps) env)))

```

master Why? Only the order of *evaluation* matters. Once the expressions have been evaluated, it literally doesn't matter in what order `cons` evaluates its arguments because they have already been evaluated and thus evaluating from left-to-right and right-to-left are equivalent. Right?

krubar

```

;; left-to-right
(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (let ((first (eval (first-operand exps) env)))
        (cons first
              (list-of-values
                (rest-operands exps)
                env)))))

;; right-to-left
(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (let ((rest (list-of-values
                   (rest-operands exps)
                   env)))
        (cons (eval (first-operand exps) env)
              rest))))

```

# sicp-ex-4.2

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.1) | Index | Next exercise (4.3) >>

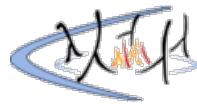
meteorgan

;; a Assignment expressions are technically pairs and will be evaluated as applications.  
Evaluating an assignment as an application will cause the evaluator to try to evaluate the  
assignment variable instead of treating it as a symbol.

```
;; b
(define (application? exp) (tagged-list? exp 'call))
(define (operator exp) (cadr exp))
(define (operands exp) (cddr exp))
```

Last modified : 2016-09-30 20:04:04  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-4.3



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.2) | Index | Next exercise (4.4) >>

meteorgan

```
(define operation-table make-table)
(define get (operation-table 'lookup-proc))
(define put (operation-table 'insert-proc))

(put 'op 'quote text-of-quotation)
(put 'op 'set! eval-assignment)
(put 'op 'define eval-definition)
(put 'op 'if eval-if)
(put 'op 'lambda (lambda (x y)
    (make-procedure (lambda-parameters x) (lambda-body x) y)))
(put 'op 'begin (lambda (x y)
    (eval-sequence (begin-sequence x) y)))
(put 'op 'cond (lambda (x y)
    (evaln (cond->if x) y)))

(define (evaln expr env)
  (cond ((self-evaluating? expr) expr)
        ((variable? expr) (lookup-variable-value expr env))
        ((get 'op (car expr)) (applyn (get 'op (car expr) expr env)))
        ((application? expr)
         (applyn (evaln (operator expr) env)
                (list-of-values (operands expr) env)))
        (else (error "Unknown expression type -- EVAL" expr))))
```

BE

One fix to the solution by meteorgan: applyn is not necessary.

```
(define (evaln expr env)
  (cond ((self-evaluating? expr) expr)
        ((variable? expr) (lookup-variable-value expr env))
        ((get 'op (car expr)) (get 'op (car expr) expr env))
        ((application? expr)
         (evaln (operator expr) env)
         (list-of-values (operands expr) env)))
        (else (error "Unknown expression type -- EVAL" expr))))
```

}}}

eric4brs

Fix to BE fix of meteorgan. Missing parens on line 4. Using BE fix as shown caused: ;The procedure #[compound-procedure 13 lookup] has been called with 4 arguments; it requires exactly 2 arguments.

```
(define (eval expr env)
  (cond ((self-evaluating? expr) expr)
        ((variable? expr) (lookup-variable-value expr env))
        ((get 'op (operator expr)) ((get 'op (operator expr)) expr env))
        ((application? expr)
         (apply (eval (operator expr) env)
                (list-of-values (operands expr) env)))
        (else (error "Unknown expression type -- EVAL" expr))))
```

bagratte

a solution using association list (one-dimensional table)

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
```

```

;; eval-rules is an association list (1d table) of
;; 'expression type'-'evaluation rule' pairs.
;; expression type is a symbol ('quote, 'define, 'lambda etc.)
;; evaluation rule must be a procedure of two arguments, exp and env.
;; defined at the end of file.
((assq (car exp) eval-rules) => (lambda (type-rule-pair)
                                         ((cdr type-rule-pair) exp env)))
 ((application? exp)
  (apply (eval (operator exp) env)
         (list-of-values (operands exp) env)))
 (else
  (error "Unknown expression type -- EVAL" exp)))

(define eval-rules
  (list (cons 'quote (lambda (exp env) (text-of-quotation exp)))
        (cons 'set! eval-assignment)
        (cons 'define eval-definition)
        (cons 'if eval-if)
        (cons 'lambda (lambda (exp env)
                           (make-procedure (lambda-parameters exp)
                                           (lambda-body exp)
                                           env)))
        (cons 'begin (lambda (exp env) (eval-sequence (begin-actions exp) env)))
        (cons 'cond (lambda (exp env) (eval (cond->if exp) env))))))

```

Sphinxsky

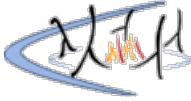
```

(define (install-syntax)
  (put 'eval 'quote
       (lambda (exp- env)
         (text-of-quotation exp-)))
  (put 'eval 'set!-
       eval-assignment)
  (put 'eval 'lambda-
       (lambda (exp- env)
         (make-procedure
           (lambda-parameters exp-)
           (lambda-body exp-)
           env)))
  (put 'eval 'define-
       eval-definition)
  (put 'eval 'if-
       eval-if)
  (put 'eval 'begin-
       (lambda (exp- env)
         (eval-sequence (begin-actions exp-) env)))
  (put 'eval 'call
       (lambda (exp- env)
         (apply-
           (eval- (operator exp-) env)
           (list-of-values (operands exp-) env))))
  (put 'eval 'cond-
       (lambda (exp- env)
         (eval- (cond->if exp-) env)))
  'ok)

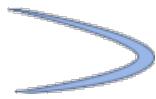
(install-syntax)

(define (eval- exp- env)
  (cond ((self-evaluating? exp-) exp-)
        ((variable? exp-) (lookup-variable-value exp- env))
        (else
          (let ((op (get 'eval (car exp-))))
            (if op
                (op exp- env)
                (error "Unknown expression type -- EVAL" exp-))))))

```



sicp-ex-4.4



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

[<< Previous exercise \(4.3\) | Index | Next exercise \(4.5\) >>](#)

woofy

```

; special forms
(define (and? exp) (tagged-list? exp 'and))
(define (and-predicates exp) (cdr exp))
(define (first-predicate seq) (car seq))
(define (rest-predicates seq) (cdr seq))
(define (no-predicate? seq) (null? seq))
(define (eval-and-predicates exps env)
  (cond ((no-predicates? exps) true)
        ((not (true? (eval (first-predicate exps)))) false)
        (else (eval-and-predicate (rest-predicates exps) env)))))

(define (or? exp) (tagged-list? exp 'or))
(define (or-predicates exp) (cdr exp))
(define (eval-or-predicates exps env)
  (cond ((no-predicates? exps) false)
        ((true? (eval (first-predicate exps))) true)
        (else (eval-or-predicate (rest-predicates exps) env)))))

; derived expressions
(define (and->if exp)
  (expand-and-predicates (and-predicates exp)))
(define (expand-and-predicates predicates)
  (if (no-predicates? predicates)
      'true
      (make-if (first-predicate predicates)
                (expand-predicates (rest-predicates predicates))
                'false)))

(define (or->if exp)
  (expand-or-predicates (or-predicates exp)))
(define (expand-or-predicates predicates)
  (if (no-predicate? predicates)
      'false
      (make-if (first-predicate predicates)
                'true
                (expand-predicates (rest-predicates predicates)))))


```

b.

```
; ; (and (list? '()) (number? 2) 3)
; ; derived into "if"
; ; (if (list? '())
; ;     (if (number? 2)
; ;         3
; ;         #f)
; ;     #f)

(define (and->if exp)
  (expand-and-clauses (and-clauses exp)))

(define (expand-and-clauses clauses)
  (cond ((empty-exp? clauses) 'false)
        ((last-exp? clauses) (first-exp clauses))
        (else (make-if (first-exp clauses)
                      (expand-and-clauses (rest-exp clauses))
                      #f)))))

(define (or->if exp)
  (expand-or-clauses (or-clauses exp)))

(define (expand-or-clauses clauses)
  (cond ((empty-exp? clauses) 'false)
        ((last-exp? clauses) (first-exp clauses))
        (else (make-if (first-exp clauses)
                      #t
                      (expand-or-clauses (rest-exp clauses))))))
```

aos

```
(define (and? exp)
  (tagged-list? exp 'and))
(define (and-expressions exp) (cadr exp))
(define (first-expression exps) (car exps))
(define (rest-expressions exps) (cdr exps))
(define (and-eval-exps exps env)
  (cond ((null? exps) 'true)
        ((null? (rest-expressions exps))
         (eval (first-expression exps) env))
        ((true? (eval (first-expression exps) env))
         (and-eval-expss (rest-expressions exps) env))
        (else 'false)))

(and-eval-expss (and-expressions exp) env)

(define (or? exp)
  (tagged-list? exp 'or))
(define (or-expressions exp) (cadr exp))
(define (or-eval-expss exps env)
  (cond ((null? exps) 'false)
        ((true? (eval (first-expression exps) env)) 'true)
        (else
         (or-eval-expss (rest-expressions exps) env)))))

(or-eval-expss (or-expressions exp) env)
```

Sticking to the book's use of 'false and 'true rather than explicitly assigning it as a boolean.

o3o3o

```
(define (and? exp)
  (tagged-list? exp 'and))

(define (or? exp)
  (tagged-list? exp 'or))

(define (eval-and exp env)
  (cond ((no-operands? exp) true)
        ((eq? false (eval (first-operand exp) env)) false)
        (else
         (eval-and (rest-operands exp) env)))))

(define (eval-or exp env)
  (cond ((no-operands? exp) false)
        ((eq? (eval (first-operand exp) env) true) true)
        (else
         (eval-or (rest-operands exp) env))))
```

x3v

For the "and" part, some of the answers above don't return the value of the last expression if all expressions evaluate to true. For example, evaluating (and 1 2) should return 2. Similar case for "or". Test cases are provided below.

Note: In scheme, only "the explicit false" object will be evaluated to false, everything else evaluates to true. See Ch 4.1.3.

```
; helper functions to make my life easier
(define (true? x) (not (eq? x false)))
(define (false? x) (eq? x false))
(define false #f)
(define test-env user-initial-environment)

(define (and? exp) (tagged-list exp 'and))
(define (and-preds exp) (cdr exp))
(define (first-pred pred-seq) (car pred-seq))
(define (rest-preds pred-seq) (cdr pred-seq))
(define (no-preds? pred-seq) (eq? pred-seq '()))
(define (eval-and-preds pred-seq env)
  (let ((val (eval (first-pred pred-seq) env)))
    (cond ((no-preds? (rest-preds pred-seq)) val)
          ((not (true? val)) 'false)
          (else (eval-and-preds (rest-preds pred-seq) env)))))
(define (eval-and exp env)
  (let ((pred-seq (and-preds exp)))
    (if (no-preds? pred-seq)
        'true
        (eval-and-preds pred-seq env)))

;; test and
(eval-and '(and 1 2) user-initial-environment) ;; 2
(eval-and '(and false 2) user-initial-environment) ;; false

(define (or? exp) (tagged-list exp 'or))
(define (or-preds exp) (cdr exp))
(define (eval-or-preds pred-seq env)
  (let ((val (eval (first-pred pred-seq) env)))
    (cond ((no-preds? (rest-preds pred-seq)) val)
          ((true? val) val)
          (else (eval-or-preds (rest-preds pred-seq) env)))))
(define (eval-or exp env)
  (let ((pred-seq (or-preds exp)))
    (if (no-preds? pred-seq)
        'false
        (eval-or-preds pred-seq env)))

;; test or
(eval-or '(or 1 2) user-initial-environment) ;; 1
(eval-or '(or false 2) user-initial-environment) ;; 2
(eval-or '(or false false) user-initial-environment) ;; #f
```

krubar

; as derived expressions

```
(define (eval-and exp env)
  (eval (and->if exp) env))

(define (eval-or exp env)
  (eval (or->if exp) env))

(define (make-if predicate
                 consequent
                 alternative)
  (list 'if
        predicate
        consequent
        alternative))

(define (and->if exp)
  (expand-and (cdr exp)))
(define (expand-and terms)
  (if (null? terms)
      #t
      (let ((first (car terms)))
```

```

        (rest (cdr terms)))
(if (null? rest)
    (make-if first first #f)
    (make-if first (expand-and rest) #f)))))

(define (or->if exp)
  (expand-or (cdr exp)))
(define (expand-or terms)
  (if (null? terms)
      #f
      (let ((first (car terms))
            (rest (cdr terms)))
        (make-if first
                  first
                  (expand-or rest)))))

(define env (null-environment 5))

(eval-or '(or #f 2) env)      ;;= -> 2
(eval-or '(or #f) env)        ;;= -> #f
(eval-or '(or) env)          ;;= -> #f

(eval-and '(and 1 2) env)    ;;= -> 2
(eval-and '(and #f 2) env)   ;;= -> #f
(eval-and '(and) env)        ;;= -> #t

```

Shade

For the derived or-expression, woofy's solution doesn't return the actual value and krubar's solution evaluates the true value twice, which is incorrect for a language with assignment.

Since we cannot explicitly evaluate the expression inside our local language's let-expression, as it makes each expression to be evaluated during expansion, we need to emulate it with the provided application and lambda abstractions:

```

(define (or->if exp) (expand-or (or-expss exp)))
(define (expand-or expss)
  (cond ((null? expss) 'false)
        ((last-exp? expss) (first-exp expss))
        (else (make-application
                  (make-lambda '(e)
                               (make-if 'e 'e (expand-or (rest-expss expss))))
                  (first-exp (car expss))))))

```

# sicp-ex-4.5

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.4) | Index | Next exercise (4.6) >>

Unknown

```
(define (eval-cond exp env)
  (let ((clauses (cdr exp)))
    (predicate car)
    (consequent cdr))
  (define (imply-clause? clause) (eq? (cadr clause) '=>))
  (define (else-clause? clause) (eq? (car clause) 'else))
  (define (rec-eval clauses)
    (if (null? clauses) 'false; checked all, no else-clause
        (let ((first-clause (car clauses)))
          (cond ((else-clause? first-clause) (eval-sequence (consequent first-clause)
env))
            ((imply-clause? first-clause) (let ((evaluated (eval (predicate first-
clause) env)))
              (if (true? evaluated)
                  (apply (eval (caddr first-clause)
env)
                    (list evaluated))
                  'false)))
            (else (if (true? (eval (predicate first-clause) env))
              (eval-sequence (consequent first-clause) env)
              'false))))))
      (rec-eval clauses)))
```

lockywolf

This procedure works, but is not compatible with the SICP's definition of eval:

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
    ((variable? exp) (lookup-variable-value exp env))
    ((quoted? exp) (text-of-quotation exp))
    ((assignment? exp) (eval-assignment exp env))
    ((definition? exp) (eval-definition exp env))
    ((if? exp) (eval-if exp env))
    ((lambda? exp)
      (make-procedure (lambda-parameters exp)
        (lambda-body exp)
        env))
    ((begin? exp)
      (eval-sequence (begin-actions exp) env))
    ((cond? exp) (eval (cond->if exp) env))
    ((application? exp)
      (apply (eval (operator exp) env)
        (list-of-values (operands exp) env)))
    (else
      (error "Unknown expression type -- EVAL" exp))))
```

That is, it doesn't transform cond syntactically.

dzy

but we can change the dispatch process in eval.. it's not a big deal.

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
    ((variable? exp) (lookup-variable-value exp env))
    ((quoted? exp) (text-of-quotation exp))
    ((assignment? exp) (eval-assignment exp env))
    ((definition? exp) (eval-definition exp env))
    ((if? exp) (eval-if exp env))
    ((lambda? exp)
      (make-procedure (lambda-parameters exp)
        (lambda-body exp)
        env))
    ((begin? exp)
      (eval-sequence (begin-actions exp) env))
    ;((cond? exp) (eval (cond->if exp) env))
    ((cond? exp) (eval-cond exp env))
    ((and? exp) (eval (and->if exp) env))
    ((or? exp) (eval (or->if exp) env))
    ((application? exp)
```

```

    (apply (eval (operator exp) env)
           (list-of-values (operands exp) env)))
  (else
    (error "Unknown expression type -- EVAL" exp)))

```

mazj

Unknown's answer may missing the part of recursively calling rec-eval.

```

(define (eval-cond exp env)
  (let ((clauses (cdr exp)))
    (predicate car)
    (consequent cdr))
  (define (imply-clause? clause) (eq? (cadr clause) '=>))
  (define (else-clause? clause) (eq? (car clause) 'else))
  (define (rec-eval clauses)
    (if (null? clauses) 'false; checked all, no else-clause
        (let ((first-clause (car clauses)))
          (cond ((else-clause? first-clause) (eval-sequence (consequent first-
clauses) env))
                ((imply-clause? first-clause) (let ((evaluated (eval (predicate
first-clause) env)))
                                         (if (true? evaluated)
                                             (apply (eval (caddr first-
clause) env)
                                                   (list evaluated))
                                             'false)))
                (else (if (true? (eval (predicate first-clause) env))
                           (eval-sequence (consequent first-clause) env)
                           'false)))
            (rec-eval (cdr clauses))))))
  (rec-eval clauses)))

```

aos

My approach just changes the way `expand-clauses` is handled. I don't do any of the evaluation and just create a separate `if` clause with its own expression.

```

(define (expand-clauses clauses env)
  (if (null? clauses)
      'false ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp
                  (cond-actions first))
                (error "ELSE clauses isn't
                      last: COND->IF"
                      clauses))
            (if (eq? (car (cond-actions first)) '=>) ; ----- here
                (make-if (cond-predicate first)
                  (list (cadr (cond-actions first))
                        (cond-predicate first))
                  (expand-clauses
                    rest
                    env))
                (make-if (cond-predicate first)
                  (cond-actions first)
                  (expand-clauses
                    rest
                    env)))))))

```

lockywolf

This won't work, because `(cond-predicate first)` would be evaluated twice, and not necessarily to the same value.

davl

It's neater to just revise cond-actions into

```

(define (cond-actions clause)
  (if (eq? '=> (cadr clause))
      (list (list (caddr clause) (cond-predicate clause)))
      (cdr clause)))

```

lockywolf

This will also evaluate the predicate twice.

zxyMike93

Here's my version of expand-cond. Since the book says the procedure after => invoke on \*the value\* of the <test>, I eval it before making it an expression.

```
(define (eval-cond exp env)
  (define (cond->if exp)
    (expand-cond (cdr exp)))
  (define (expand-cond clauses)
    (cond [(null? clauses) #f]
          [else
            (let ([first (car clauses)] [rest (cdr clauses)])
              (cond [(eq? 'else (car first))
                     (if (null? rest)
                         (sequence->exp (cdr first))
                         (error "Clauses after else"))]
                    [else
                      (if (eq? '=> (cadr first))
                          ; action for =>
                          (make-if (car first)
                                   (sequence->exp (cons (caddr first)
                                                         (meta-eval (car first) env)))
                                   (expand-cond rest))
                          (make-if (car first)
                                   (sequence->exp (cdr first))
                                   (expand-cond rest))))]))]
            (meta-eval (cond->if exp) env)))
```

And I test it by evaling the book example. However, quote(') it's a bit confusing...

```
(eval '(cond ((assoc 'b ('(a 1) ('b 2))) => cadr) (else false)) global-env)
```

PS: Sorry for the [], I'm using Racket here.

RaphyJake

This is my version of expand-clauses. It turns the special cond syntax into a lambda expression, so the condition gets evaluated only once.

```
(define (cond-clauses exp) (cdr exp))
(define (cond-else-clause? clause) (eq? (cond-predicate clause) 'else))

(define (cond-predicate clause) (car clause))
(define (cond-special-syntax? clause) (eq? (cadr clause) '=>))
(define (cond-special-syntax-function clause) (caddr clause))
(define (cond-actions clause) (cdr clause))

(define (cond->if exp)
  (expand-clauses (cond-clauses exp)))

(define (expand-clauses clauses)
  (if (null? clauses)
      'false ; no else clause
      (let ([first (car clauses)]
            [rest (cdr clauses)])
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp
                 (cond-actions first))
                (display "ELSE clause isn't last: COND->IF"))
            (if (cond-special-syntax? first)
                (list (make-lambda (list 'x)
                                  (list (make-if 'x
                                                (list (cond-special-syntax-function first)
                                                      (expand-clauses rest)))))))
                (cond-predicate first))
            (make-if (cond-predicate first)
                     (sequence->exp
                      (cond-actions first))
                     (expand-clauses
                      rest)))))))
```

The problem with Ralphy Jake's solution is it introduces an undeclared variable x to the environment and can change the meaning of expressions that depend on the value of x in the scope outside of the cond clause. Try the following 2 tests and you will see the first gives an unexpected result because of the introduction of the hidden 'x'.

```
(define x 10)

(define (test1)
  (cond ((assoc 'a '((a 1) (b 2))) => (lambda (y) (display y) (display x)))
        (else 'never)))

(test1)
(a 1)(a 1)
#<void>
```

Here the answer is the result of the assoc list test and not the original value of x

```
(define y 10)

(define (test2)
  (cond ((assoc 'a '((a 1) (b 2))) => (lambda (z) (display z) (display y)))
        (else 'never)))

(test2)
(a 1)10
#<void>
```

Here we get 10, the correct answer. We are lucky that we didn't pick an identifier that the interpreter hides!

master

I'm not very confident in my solution.

```
(define (expand-clauses clauses)
  (if (null? clauses)
      'false ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last: COND->IF"
                      clauses))
            (if (arrow? first)
                (arrow->exp first)
                (make-if (cond-predicate first)
                         (sequence->exp (cond-actions first))
                         (expand-clauses rest))))))

(define (arrow? clause) (eq? (cadr clause) '=>))
(define (arrow->exp clause) (let ((test (cond-predicate clause)))
                                (recipient (caddr clause)))
                                (make-if test
                                         (recipient test)
                                         'false)))
```

x3v

Environment for testing: user-initial-environment.

I ended up writing an eval-cond procedure, added extra env arg to cond-if, expand-clauses, and make-if, such that double evaluation of the predicate expression can be avoided. Main change is in the make-if procedure.

```
(define (eval-cond exp env)
  (eval (cond->if exp env) env))
(define (cond? exp) (tagged-list? exp 'cond))
(define (cond-clauses exp) (cdr exp))
(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))
(define (cond-predicate clause) (car clause))
(define (cond-actions clause) (cdr clause))
(define (cond->if exp env) ; added env arg
  (expand-clauses (cond-clauses exp) env))

(define (expand-clauses clauses env) ; added env arg
  (if (null? clauses)
      'false
      (let ((first (car clauses)))
```

```

        (rest (cdr clauses)))
(if (cond-else-clause? first)
  (if (null? rest)
      (sequence->exp (cond-actions first))
      (error "ELSE clause isn't last -- COND->IF"
             clauses))
  (make-if (cond-predicate first)
           (sequence->exp (cond-actions first))
           (expand-clauses rest env)
           env)))))

;; main change is here
(define (make-if predicate consequent alternative env)
  (if (eq? (cadr consequent) '=>)
      (let ((value (eval predicate env)))
        (list 'if value (lambda () ((caddr consequent) value)) alternative))
      (list 'if predicate consequent alternative)))

(eval '(cond ((assoc 'b '((a 1) (b 2))) => cadr) (else false)) test-env) ;; 2

```

ce

I believe that with this solution, every single cond recipient clause will have its predicate evaluated, since expand-clauses will construct the entire cond expression. So, if any of these predicate expressions have side effects, the program is likely to behave incorrectly (and probably in a way that is very perplexing to debug); the proper behavior would be to only evaluate a predicate if every earlier predicate had already failed. For that reason, I suspect that sneaking an eval into cond->if or any of its dependencies is probably never going to be the right approach.

seninha

My solution requires the implementation of a `make-let` procedure, which basically creates the application (created by a new `make-application` procedure) of a lambda (created by the existing `make-lambda` procedure) to a given value.

Using `make-let` there's no need to pass the environment to `expand-clauses` as a new argument.

```

(define make-application
  (lambda (operator operands)
    (cons operator operands)))

(define make-let
  (lambda (var val body)
    (make-application (make-lambda (list var) body) val)))

(define cond-recipient-clause?
  (lambda (clause)
    (eq? (car (cond-actions clause)) '=>)))

(define cond-recipient
  (lambda (clause)
    (cadr (cond-actions clause)))))

(define expand-clauses
  (lambda (clauses)
    (if (null? clauses)
        'false
        ; no else clause
        (let ((first (car clauses))
              (rest (cdr clauses)))
          (cond
            ((cond-else-clause? first)
             (if (null? rest)
                 (sequence->exp (cond-actions first))
                 (error "ELSE clause isn't last -- COND->IF" clauses)))
            ((cond-recipient-clause? first)
             (make-let 'result
                       (cond-predicate first)
                       (make-if 'result
                                 (make-application (cond-recipient first)
                                                 (list 'result))
                                 (expand-clauses rest))))
            (else
              (make-if (cond-predicate first)
                      (sequence->exp (cond-actions first))
                      (expand-clauses rest))))))))))

```

hi

evaluated only once no naming conflicts

```

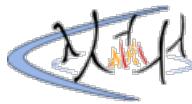
((cond-apply-clause? clause)
 (make-application

```

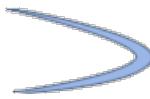
```
(make-lambda '(pred consequence alternetive)
             (list (make-if 'pred
                           (list (make-application 'consequence nil)
                                 (make-application 'alternetive nil))))
                   (list (clause-predicate clause)
                         (make-lambda '()
                           (list (clause-procedure clause)))
                         (make-lambda '()
                           (list (clauses->if (rest-clauses clauses)))))))
```

---

Last modified : 2023-02-16 07:22:06  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-4.6



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.5) | Index | Next exercise (4.7) >>

meteorgan

```
;; in eval, add this:  
((let? expr) (evaln (let->combination expr) env))  
  
;; let expression  
(define (let? expr) (tagged-list? expr 'let))  
(define (let-vars expr) (map car (cadr expr)))  
(define (let-init expr) (map cadr (cadr expr)))  
(define (let-body expr) (cddr expr))  
  
(define (let->combination expr)  
  (list (make-lambda (let-vars expr) (let-body expr))  
        (let-init expr)))
```

Hertz

Because the initial values for the vars serve as the remainder of the list (cdr) instead of a separate list, the let->combination should be defined using 'cons' instead of 'list'.

```
(define (let->combination expr)  
  (cons (make-lambda (let-vars expr) (let-body expr))  
        (let-init expr)))
```

Dolemo

I think let-body above seems to have a bug.

```
(define (let-body expr) (caddr expr))
```

because body part is one of three parts of let expression list.

Shi-Chao

Body part may contain several expressions, so 'cddr' is used to point all the rest parts.

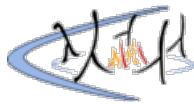
krubar

```
;; without using map  
(define (let-bindings exp)  
  (cadr exp))  
(define (let-body exp)  
  (cddr exp))  
  
(define (bindings->params bindings)  
  (if (null? bindings)  
      bindings  
      (cons  
        (caar bindings)  
        (bindings->params (cdr bindings)))))  
  
(define (bindings->args bindings)  
  (if (null? bindings)  
      bindings  
      (cons  
        (cadar bindings)  
        (bindings->args (cdr bindings)))))
```

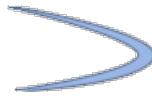
```
;; TODO Check if bindings are pairs
(define (let->combination exp)
  (cons
    (make-lambda
      (bindings->params (let-bindings exp))
      (let-body exp))
    (bindings->args (let-bindings exp))))
```

pvk

This is extremely nitpicky, but I'd prefer to use make-application than cons here, since it maintains the abstraction barrier of the implementation of procedure application.



# sicp-ex-4.7



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.6) | Index | Next exercise (4.8) >>

meteorgan

```
; let* expression
(define (let*-? expr) (tagged-list? expr 'let*))
(define (let*-body expr) (caddr expr))
(define (let*-inits expr) (cadr expr))
(define (let*->nested-lets expr)
  (let ((inits (let*-inits expr))
        (body (let*-body expr)))
    (define (make-lets exprs)
      (if (null? exprs)
          body
          (list 'let (list (car exprs)) (make-lets (cdr exprs))))))
    (make-lets inits)))
```

3pmtea

The body of let\* is a sequence of expressions, so I think it should be accessed with cddr. It can be transformed to a single expression with sequence->exp which is defined in the book.

```
(define (let-args exp) (cadr exp))
(define (let-body exp) (cddr exp))
(define (make-let args body) (cons 'let (cons args body)))

(define (let*->nested-lets exp)
  (define (reduce-let* args body)
    (if (null? args)
        (sequence->exp body)
        (make-let (list (car args))
                 (list (reduce-let* (cdr args) body)))))

  (reduce-let* (let-args exp) (let-body exp)))
```

test:

```
> (let*->nested-lets '(let* ((x 1) (y 2)) x y))
'(let ((x 1)) (let ((y 2)) (begin x y)))
> (let*->nested-lets '(let* () 1))
1
```

mazj

Compared with 3pmtea's answer, the let expression generated in my solution is more elegant, but the implementation is worse.

```
(define (let*->nested-lets exp env)
  (define (make-let var-pairs body-exp)
    (cons 'let (cons var-pairs body-exp)))
  (define (nested-lets var-pairs body-exp)
    (if (null? var-pairs) body-exp
        (make-let (list (car var-pairs))
                  (if (null? (cdr var-pairs))
                      (nested-lets (cdr var-pairs) body-exp)
                      (list (nested-lets (cdr var-pairs) body-exp))))))

  (if (null? (cadr exp)) (cons 'begin (cddr exp))
      (nested-lets (cadr exp) (cddr exp))))
```

test:

```
> (let*->nested-lets '(let* ((x 1) (y 2)) x y) 233)
(let ([x 1]) (let ([y 2]) x y))
> (let*->nested-lets '(let* () x y) 233)
(begin x y)
```

inchmeal

Another solution using fold:

```
;; var-defs and body should be list
(define (make-let var-defs body)
  (cons 'let (cons var-defs body)))
(define (let*-? exp) (tagged-list? exp 'let*))
(define (let*->let exp)
  (car
    (fold-right (lambda (new rem)
                  (list (make-let (list new) rem)))
                (let-body exp)
                (let-vardefs exp))))
```

someone

Simpler:

```
(define (let*->nested-lets exp)
  (define (wrap def exp)
    (list 'let (list def) exp))
  (let ((bindings (cadr exp))
        (body (caddr exp)))
    (fold-right wrap body bindings)))
```

dzy

```
(define (let*-? exp) (tagged-list? exp 'let*))
(define (let*->nested-lets exp)
  (expand-let*-clauses (cadr exp) (cddr exp)))
(define (expand-let*-clauses lets body)
  (if (null? lets)
      (sequence->exp body)
      (list 'let (list (car lets)) (expand-let*-clauses (cdr lets)
body))))
```

master

Here's my attempt at answering the question posed by the authors in the second part of the question. The answer becomes a little clearer if we use `let->combination` to expand the expression produced by `let*->nested-lets`:

```
(let->combination (let*->nested-lets '(let* ((x 1) (y (+ x 1))) (+ x y)))
;; '(call (lambda (x) (let ((y (+ x 1))) (+ x y))) 1)
```

The problem is that `let->combination` doesn't recursively expand the nested `lets`.

Although not the only way to implement it in our evaluator, the most straightforward way is as follows:

```
(define (eval exp env)
;; ...
  ((application? exp)
   (apply (eval (operator exp) env)
         (list-of-values (operands exp) env)))
;; ...
  ((lambda? exp) (make-procedure (lambda-parameters exp)
                                 (lambda-body exp)
                                 env)))
;; ...
  ((let? exp) (eval (let->combination exp) env)))
;; ...
```

So, when we ask `eval` to evaluate `'(call (lambda (x) (let ((y (+ x 1))) (+ x y))) 1)`, it will recognize it as a procedure application and try to apply `(lambda (x) (let ((y (+ x 1))) (+ x y)))` to 1. Will this work? That depends on the implementation of `apply`. It uses `eval-sequence` to apply compound procedures to arguments, so if `apply` succeeds in fully simplifying this expression then the expression will be evaluated correctly. However, it would make more sense to rewrite `let->combination` so that it can handle nested `lets` instead of relying on the caller knowing that it needs to be expanded again.

```
(define (let->combination exp)
  (define (rec this-exp)
    (let ((next-exp (caddr this-exp))))
```

```

(if (let? next-exp)
    (make-lambda (let-vars this-exp)
        (list (make-application (rec next-exp) (let-exp next-exp))))
    (make-lambda (let-vars this-exp)
        (let-body this-exp))))
(make-application (rec exp) (let-exps exp)))

```

This procedure produces expressions of the form:

```

(let->combination (let*->nested-lets ' (let* ((x 1) (y (+ x 1))) (+ x y)))
;; '(call (lambda (x) (call (lambda (y) (+ x y)) (+ x 1))) 1)

```

Should be no problem for our evaluator.

pvk

I'm confused by this comment. Is there an example of how the evaluator could fail on the first version of the expression?

Unraveling the definition of `apply` as given so far in the book, we're going to

```

(eval-sequence (procedure-body procedure)
  (extend-environment (procedure-parameters procedure)
    arguments
    (procedure-environment procedure)))

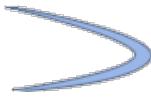
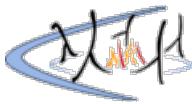
```

where

- (procedure-body procedure) is `((let ((y (+ x 1))) (+ x y)),
- (procedure-parameters procedure) is `(x),
- and arguments is `(1)
- (and (procedure-environment procedure)) is the environment where the original `let*` expression was evaluated). Since the procedure body has a single expression, `eval-sequence` reduces to `eval`. In other words, we're going to evaluate the inner `let` expression in an environment extended by the definition that `x = 1`, which is exactly what we should be doing.

So I don't think it makes a difference whether we immediately translate the `let*` expression to nested `lambdas`, or translate it to `nested-lets` and let the interpreter of `let` finish the job. Am I missing something?

# sicp-ex-4.8



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.7) | Index | Next exercise (4.9) >>

meteorgan

```
(define (named-let? expr) (and (let? expr) (symbol? (cadr expr))))  
  
(define (named-let-func-name expr) (cadr expr))  
  
(define (named-let-func-body expr) (caddr expr))  
  
(define (named-let-func-parameters expr) (map car (caddr expr)))  
  
(define (named-let-func-initns expr) (map cadr (caddr expr)))  
  
(define (named-let->func expr)  
  (list 'define  
        (cons (named-let-func-name expr) (named-let-func-parameters expr))  
        (named-let-func-body expr)))  
  
(define (let->combination expr)  
  (if (named-let? expr)  
      (sequence->exp  
        (list (named-let->func expr)  
              (cons (named-let-func-name expr) (named-let-func-initns expr)))  
        (cons (make-lambda (let-vars expr)  
                           (list (let-body expr)))  
              (let-initns expr))))
```

karthikk

While the above should work, the problem with doing it with **define** is that it raises the possibility of nameclash issues, as define directly installs the name of the lambda in the named let into the current frame of the environment. The other possibility, which allows greater control of scope is to do it with the usual let, adding an arbitrary binding for var which is then reassigned in the body of the let expression (with a set! command) to the needed lambda before anything else is evaluated...

inchmeal

One approach where name of procedure is only available inside the body:

```
(define (let? exp) (tagged-list? exp 'let))  
(define (let-has-name? exp) (symbol? (cadr exp)))  
(define (let-name exp) (cadr exp))  
(define (let-vardefs exp)  
  (if (let-has-name? exp)  
      (caddr exp)  
      (cadr exp)))  
  
(define (let-body exp)  
  (if (let-has-name? exp)  
      (cdddr exp)  
      (cddr exp)))  
  
(define (let->combination exp)  
  (let ((res (fold-right  
             (lambda (new rem)  
               (cons (cons (car new) (car rem))  
                     (cons (cadr new) (cdr rem))))  
             (cons '() '())  
             (let-vardefs exp))))  
    (let ((vars (car res))  
          (vexprs (cdr res)))  
      (define proc (make-lambda vars (let-body exp)))  
      (if (let-has-name? exp)  
          ;;create a lambda with no args containing:  
          ;;(i) definition of the actual lambda(proc)  
          ;;(ii) invocation of proc with supplied expressions.  
          ;;finally create application for this no argument lambda.  
          (cons
```

```

        (make-lambda '()
                      (list (list 'define (let-name exp) proc)
                            (cons (let-name exp) vexp)
                            )))
        '())
      (cons proc vexp)
    ))))

```

3pmtea

I'm with this approach, that the effect of scoping can be achieved by defining a lambda with no args and calling it immediately.

```

dzy
(define (let->combination exp)
  (if (variable? (cadr exp))
      (cons (make-lambda '()
                          (sequence->exp (list 'define
                                                (cadr exp) (make-lambda (map car (caddr exp)
                                                               (sequence->exp (cdddr exp)))))))
            (list (cadr exp) (map cdr (caddr exp))))))
      '())
  (cons (make-lambda (let-parameter exp)
                     (caddr exp))
        (let-arguments exp)))

```

Sphinxsky

```

; ======expression=====
; n >= 0
(define (expression-data exp- n)
  (if (and (number? n) (>= n 0))
      (list-ref exp- n)
      (error "Error parameter of n -- EXPRESSION-DATA!")))

(define (expression-tag exp-)
  (expression-data exp- 0))

(define (exp-data-after-n exp- n)
  (if (= n 0)
      exp-
      (exp-data-after-n (cdr exp-) (- n 1)))))

(define (tagged-expression? exp- tag)
  (if (pair? exp-)
      (eq? (expression-tag exp-) tag)
      false))

; ======let=====

(define (let-name exp-)
  (let ((name (expression-data exp- 1)))
    (if (variable? name)
        name
        #f)))

(define (let-body exp-)
  (exp-data-after-n
   exp-
   (if (let-name exp-) 3 2)))

(define (let-variables exp-)
  (expression-data
   exp-
   (if (let-name exp-) 2 1)))

(define (consortium-variable consortium)
  (expression-data consortium 0))

(define (consortium-value consortium)
  (expression-data consortium 1))

```

```

(define (separate variables)
  (define (iter variables variable value)
    (if (null? variables)
        (cons (reverse variable) (reverse value))
        (let ((first (car variables)))
          (iter
            (cdr variables)
            (cons (consortium-variable first) variable)
            (cons (consortium-value first) value)))))

  (iter variables '() '()))

(define (is-let? exp)
  (tagged-expression? exp- 'let-))

(define (make-let bindings body)
  (cons
    'let-
    (cons
      bindings
      (if (is-let? body)
          (list body)
          body)))))

(define (let->combination exp-)
  (let* ((name (let-name exp-))
         (var (separate (let-variables exp-)))
         (lambda-exp (make-lambda (car var) (let-body exp-))))
         (if name
             (make-let
               (list (list name lambda-exp))
               (list (cons 'call (cons name (cdr var)))))))
         (cons
           'call
           (cons lambda-exp (cdr var)))))

  (put 'eval 'let-
    (lambda (exp- env)
      (eval- (let->combination exp-) env)))))


```

x3v Could define named let selectors if need be.

```

(define (let->combination exp)
  (if (symbol? (cadr exp))
      (list (make-lambda (list (cadr exp)) (list (cadddr exp))) (caddr exp))
      (cons (make-lambda (let-vars exp) (let-body exp))
            (let-expss exp)))))

;; test
(eval (let->combination '(let a 0 (+ 1 a))) test-env) ;; 1

```

krubar

```

;; selectors

(define (let-bindings exp)
  (if (named-let? exp)
      (caddr exp)
      (cadr exp)))

(define (let-body exp)
  (if (named-let? exp)
      (cadddr exp)
      (caddr exp)))

(define (bindings->params bindings)
  (if (null? bindings)
      bindings
      (cons
        (caar bindings)
        (bindings->params (cdr bindings)))))

(define (bindings->args bindings)
  (if (null? bindings)
      bindings
      (cons
        (cadar bindings)
        (bindings->args (cdr bindings)))))


```

```

(define (named-let? exp)
  (not (pair? (cadr exp))))

(define (let-var exp)
  (cadr exp))

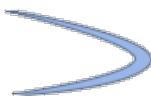
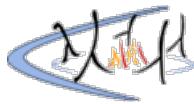
(define (let->combination exp)
  (if (named-let? exp)
      (named-let->combination exp)
      (cons (make-lambda
              (bindings->params (let-bindings exp))
              (list (let-body exp)))
            (bindings->args (let-bindings exp)))))

;; binding let var and bindings with lambdas
(define (named-let->combination exp)
  (cons
    (list
      (make-lambda '(f) '(((f f)))
      (make-lambda (list (let-var exp))
        (list (make-lambda (bindings->params (let-bindings exp))
          (list
            (list (make-lambda (list (let-var exp))
              (list (let-body exp)))
            (list (let-var exp) (let-var exp)))))))
        (bindings->args (let-bindings exp)))))

; (let->combination '(let ! ((x 5)) (if (< x 2) 1 (* (! (- x 1)) x))))
;
; -> (((lambda (f) (f f))
;       (lambda (!)
;         (lambda (x)
;           ((lambda (!)
;             (if (< x 2) 1
;               (* (! (- x 1)) x)))
;             (! !))))
;           5)
; ;
; -> 120

```

# sicp-ex-4.9



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.8) | Index | Next exercise (4.10) >>

woofy

```
; (while <predicate> <body>)
; for example:
; (while (< i 100)
;       (display i)
;       (set! i (+ i 1)))

(define (while? exp) (tagged-list? exp 'while))
(define (while-predicate exp) (cadr exp))
(define (while-body exp) (cddr exp))

(define (make-procedure-definition name parameters body)
  (cons 'define (cons (cons name parameters) body)))
(define (make-procedure-application procedure arguments)
  (cons procedure arguments))

(define (while->combination exp)

  (define (while->procedure-def procedure-name)
    (make-procedure-definition
      procedure-name
      '()
      (make-if
        (while-predicate exp)
        (sequence->exp
          (append (while-body exp)
                  (make-procedure-application procedure-name '())))))

  ; wrap the procedure definition in a lambda to constrain its scope
  (make-procedure-application
    (make-lambda
      '()
      (list (while->procedure-def 'while-procedure)
            (make-procedure-application 'while-procedure '())))
    '()))

; the whole thing will look like this:
((lambda ()
  (define (while-procedure)
    (if (< i 100)
        (begin
          (display i)
          (set! (+ i 1)
                (while-procedure))))
        (while-procedure))))
```

ce

Unless I'm mistaken, this can behave incorrectly if the evaluation environment already has an expression with the name while-procedure. Whether or not that's likely to happen, and whether or not it's easy to work around, it's probably not a good thing to let the implementation details of the evaluator leak in such a way.

meteorgan

```
; i only implement while.
;; add this to eval
((while? expr) (evaln (while->combination expr) env))

;; while expression
(define (while? expr) (tagged-list? expr 'while))
```

```

(define (while-condition expr) (cadr expr))
(define (while-body expr) (caddr expr))
(define (while->combination expr)
  (sequence->exp
    (list (list 'define
                (list 'while-iter)
                (make-if (while-condition expr)
                         (sequence->exp (list (while-
                           body expr)
                           (list 'while-iter)))))))
    'true)))

```

karthikk

Because while has been implemented by *meteorgan* as a define expression, with a single name 'while-iter', it can cause serious nameclash issues if there are two while loops in a single procedure!

The alternative is to implement while with a lambda i.e. with a let expression as below (below assumes the let macro has been installed into eval procedure):

```

(define (while? exp) (tagged-list? exp 'while))
(define (while-pred exp) (cadr exp))
(define (while-actions exp) (caddr exp))

(define (make-single-binding var val) (list (list var val)))
(define (make-if-no-alt predicate consequent) (list 'if predicate consequent))
(define (make-combination operator operands) (cons operator operands))

(define (while->rec-func exp)
  (list 'let (make-single-binding 'while-rec '(quote *unassigned*))
        (make-assignment 'while-rec
          (make-lambda '()
            (list (make-if-no-alt
                  (while-pred exp)
                  (make-begin (append (while-actions exp)
                      (list (make-combination 'while-rec
                        '()))))))))
        (make-combination 'while-rec '())))

```

djrochford

An alternative, not very clever way of dealing with the scope issue karthikk points out is to have a name to use in the procedure definition as part of the while construct -- i.e., have while syntax be something like this: `(while <name> <predicate> <body>)`. Unusual, but much cleaner.

joew

I was aware of the scope issue so I used the macro name as the define procedure name. Definitely a kludge but it actually works and you can even nest loops this way. My evaluator looks for the "while" tagged list and then I reuse while knowing the programmer can't use while as it's already a language construct.

There might be another way by just evaling each iteration without any defintion and not creating derived expressions, but I didn't bother.

```

(define (make-definition label value) (list 'define label value))

(define (while->lambda exp)
  (let ((check (cadr exp))
        (body (caddr exp)))
    (make-begin
      (list (make-definition 'while
                            (make-lambda '()
                              (list (make-if check
                                (make-begin
                                  (list body
                                        (list 'while)))
                                "done")))))
    (list 'while)))))


```

pvk

How do you make your evaluator interpret this correctly? It seems to me that if the evaluator checks for `while` statements before it tries to interpret expressions as procedure applications, it will always interpret these inner occurrences of `while` as actual `while` statements, despite the new definition.

jotti

Comparing my solution to the other ones here, I can't help but feel I may be missing something, I would be glad to hear if there are any flaws. I've implemented my `while` in terms of `if` and by recursively calling `while` again.

```
(define (while? expr)
  (tagged-list? expr 'while))

(define (while-predicate expr) (cadr expr))

(define (while-body expr) (caddr expr))

(define (eval-while expr env)
  (let ((pred (while-predicate expr))
        (body (while-body expr)))
    (eval (make-if pred (sequence->exp (list body expr)) "done") env)))
```

ce

This solution might "do the right thing," but I don't think it qualifies as a derived expression.

pvk

I currently think this is the best solution on this page, despite/because of not being derived. Every other definition of `while`, as far as I can tell, introduces a `let` or `define` which risks name collision if the body of the `while` happens to refer to variables with the same name (i.e. the same problem @ce mentioned in response to @woofy).

The ways I can see around this issue are:

1. Reserve additional keywords such as `while-rec` in the definition of the programming language;
2. Implement special syntax for `while` as done here;
3. Make the programmer specify a name for while loops (@djrochford's suggestion);
4. Make the interpreter scan the code for all variable names before translating derived expressions, and guarantee that new variable names introduced in the translation of derived expressions are distinct from any used by the programmer.

Are there other alternatives?

squarebat

I implemented for loop evaluator as a derivation of named-let in previous exercise. It makes a few brave assumptions, like an already existing implementation of make-named-let, but it is simpler to understand

```
;e.g of a for loop
(for (i 0) (< i 10) (+ i 1)
  sequence-of-exp)

;e.g of while loop
(while (< var 10) sequence)

;e.g of do while
(do ((exp1)
      (exp2)
      (...))
    (expn))
  until (< var 10))

;convert for to the following form
(let for ((i 0)
          (count 10)
          (body (sequence->exp sequence-of-exp)))
  (if (< i count)
      (begin
        (body)
        (for (+ i 1) 10 body)))))

;evaluator for for loop
;syntax checking has not been implemented for simplicity (totally not because I don't want
to)
(define (for? exp) (tagged-list? exp 'for))
(define (for-body exp) (cddddr exp))
(define (for-iter exp) (cadr exp))
(define (for-count exp) (caddr (caddr exp)))
(define (for-predicate exp) (caddr exp))
(define (for-change-iter exp) (cadddr exp))
(define (for->named-let exp)
  (make-named-let 'for
```

```

(list
  (for-iter exp)
  (cons 'count (for-count exp))
  (cons 'body (sequence->exp (for-body exp))))
(make-if (for-predicate exp)
  (make-begin '(body)
    ' (for (for-change-iter exp)
      count
      body))
  'done)))

```

x3v

Added syntax for python's list comprehensions, pretty simple tbh as it's very similar to map, so i might do one for while too.

Scheme list comprehension : `((* i i) for i in (list 1 2 3 4 5))`

Python list comprehension : `[i * i for i in [1, 2, 3, 4, 5]]`

```

(define (list-comp? exp) (and (pair? exp) (eq? (cadr exp) 'for)))
(define (list-comp-exp exp) (car exp))
(define (list-comp-var exp) (caddr exp))
(define (list-comp-iterable exp) (car (cddddr exp)))
(define (eval-list-comp exp env)
  (define (iter iterable)
    (if (eq? iterable '())
        '()
        (cons
          (eval (list
            (make-lambda
              (list (list-comp-var exp))
              (list (list-comp-exp exp)))
            (car iterable)) env)
          (iter (cdr iterable))))))
  (iter (eval (list-comp-iterable exp) env)))

(define list-comprehension-test
  '((* i i) for i in (list 1 2 3 4 5)))
  (eval list-comprehension-test test-env) ;; (1 4 9 16 25)

```

closeparen

I accidentally did this one out of order with 4.08, so I had to come up with another way to make an anonymous recursive procedure. I also leveraged "quasiquoting," which I learned about out of turn, to make my solution very small. For those unfamiliar, backtick is like quote except you can escape from it with comma.

```

(define (make-while test body) (list 'while test body))
(define (while? exp) (tagged-list? exp 'while))
(define (while-test exp) (cadr exp))
(define (while-body exp) (caddr exp))

(define (expand-while exp)
  `((let ((iter (lambda (next)
                  (if ,(while-test exp) (begin , (while-body exp) (next next)))))))
     (iter iter)))

```

krubar

```

;; doseq from clojure, implemented in terms of named let from exercise 4.8

(define (doseq? exp) (tagged-list? exp 'doseq))
(define (doseq->let exp)
  (list 'let 'doseq (list (cadr exp)))
    (list 'if (list 'pair? (car (bindings->params (list (cadr exp))))))
      (sequence->exp (list
        (list
          (make-lambda (bindings->params (list (cadr exp)))
            (list (let-body exp)))
          (list 'car (car (bindings->params (list (cadr exp)))))))
        (list 'doseq (list 'cdr (car (bindings->params (list (cadr exp)))))))))))
  ; (let->combination (doseq->let ' (doseq (x '(1 2 3 4 5)) (display x)))))

```

haha

make pressions to evaluate in a same environment, no other binds

```
(define (while->combination exp)
  (let ((predicate-exp (while-predicate exp))
        (body-expss (while-body exp)))
    (make-application
      (make-lambda (list 'pred-proc 'body-proc)
        (list
          (make-define '(f)
            (list
              (make-if (make-application 'pred-proc
                nil)
                (make-begin (list
                  (make-application 'body-proc
                    nil)
                  (make-application 'f nil)))
                ''done)))
              (make-application 'f nil)))
        (list (make-lambda '()
          (list predicate-exp))
          (make-lambda '()
            body-expss)))))

;;
print: (while->combination '(while (< n 10) (set! n (+ n 1)))))

((lambda (pred-proc body-proc) (define (f) (if (pred-proc) (begin (body-proc) (f))
'done)) (f))
  (lambda () (< n 10))
  (lambda () (set! n (+ n 1)))))
```

---

Last modified : 2023-07-18 19:58:47  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-4.10

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.9) | Index | Next exercise (4.11) >>

felix021

```
;;
;; Say, there's another syntax which places the function name at the end of a list:
;; (1 2 3 +)
;; if we change related functions such as tagged-list?, if?, eval/apply won't be affected.

(define (last-element lst)
  (if (null? (cdr lst))
      (car lst)
      (last-element (cdr lst)))))

(define (tagged-list? exp sym)
  (if (pair? exp)
      (let ((last (last-element exp)))
        (eq? last sym))
      #f))

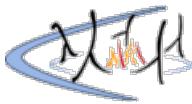
(define (if? exp) (tagged-list? exp 'if))

(define (if-predicate exp) (car exp))

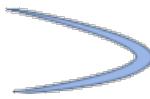
(define (if-consequent exp) (cadr exp))

(define (if-alternative exp)
  (if (= (length exp) 4)
      (caddr exp)
      'false))

; ...
```



# sicp-ex-4.11



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.10) | Index | Next exercise (4.12) >>

meteorgan

```
(define (make-frame variables values)
  (if (= (length variables) (length values))
      (map cons variables values)
      (error "length mismatch -- MAKE-FRAME" variables values)))

(define (frame-variables frame) (map car frame))
(define (frame-values frame) (map cdr frame))
```

Rptx

meteorgans answer does not work. You must change at least define-variable! and set-variable-value! for the system to work.

felix021 (a full version)

```
;;
;

;SKIP(no change):
;  enclosing-environment
;  first-frame
;  the-empty-environment

(define (make-frame variables values)
  (cons
    'table
    (map cons variables values)))

(define (frame-pairs frame) (cdr frame))

(define (add-binding-to-frame! var val frame)
  (set-cdr! frame
    (cons (cons var val) (frame-pairs frame)))))

;SKIP:
;  extend-environment

(define (lookup-variable-value var env)
  (define (env-loop env)
    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((ret (assoc var (frame-pairs (first-frame env))))))
          (if ret
              (cdr ret)
              (env-loop (enclosing-environment env))))))
  (env-loop env))

(define (set-variable-value! var val env)
  (define (env-loop env)
    (if (eq? env the-empty-environment)
        (error "Unbound variables -- SET!" var)
        (let ((ret (assoc var (frame-pairs (first-frame env))))))
          (if ret
              (set-cdr! ret val)
              (env-loop (enclosing-environment env))))))
  (env-loop env))

(define (define-variable! var val env)
  (let* ((frame (first-frame env))
         (ret (assoc var (frame-pairs frame))))
    (if ret
        (set-cdr! ret val)
        (add-binding-to-frame! var val frame))))
```

aos

Yep -- ultimately, I went with this approach too of using an "associative list". Although interestingly, attempting to use the Scheme implementation of `assoc` did not work and had to use the book's approach!

mazj

A way not use associative list:

```
(define (make-frame var val) (cons 'frame (map cons var val)))
(define (add-binding-to-frame! var val frame)
  (set-cdr! frame (cons (cons var val) (cdr frame))))
(define (set-var-to-frame! var val frame)
  (let ((find #f))
    (set-cdr! frame
      (map (lambda (pair)
        (if (eq? (car pair) var)
            (begin (set! find #t) (cons var val))
            pair))
      (cdr frame)))
    find))
(define (set-variable-value! var val env)
  (env-loop env)
  (if (eq? env the-empty-environment)
      (error "Unbound variable -- SET!" var)
      (if (set-var-to-frame! (first-frame frame)) 'done
          (env-loop (enclosing-environment env)))))
  (env-loop env))
(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (if (set-var-to-frame! var val frame) 'done
        (add-binding-to-frame! var val frame))))
```

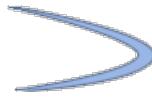
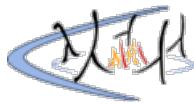
pvk

This is probably obvious to many, but only clicked for me while doing this exercise: `set!` is fundamentally different from `set-car!` and `set-cdr!`. (`set! foo val`) modifies the *binding* of the name `foo`: thus, any procedure of the form

```
(define (add-binding-to-frame! var val frame)
  (set! frame ...))
```

is useless, as it only redefines the binding of the procedure-local variable `frame`, not whatever `frame` we passed to the procedure. `set-car!` and `set-cdr!`, rather than just redefining a name, actually modify the pair that the name points to, and thus can be used in such contexts.

# sicp-ex-4.12



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.11) | Index | Next exercise (4.13) >>

woofy

not a very good example on generalization but most likely what the author intended

```
(define (tranverse var env on-find on-frame-end on-env-end)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars) (on-frame-end env))
            ((eq? var) (on-find vals))
            (else (scan (cdr vars) (cdr vals))))))
    (if (eq? env the-empty-environment)
        (on-env-end)
        (let ((frame (first-frame)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))

(define (lookup-variable-value var env)
  (tranverse var
             env
             (lambda (vals) (car vals))
             (lambda (env) (lookup-variables-value var (enclosing-environment env)))
             (lambda () (error "Unbound variable -- lookup-variable-value " var)))))

(define (set-variable-value! var val env)
  (tranverse var
             env
             (lambda (vals) (set-car! vals val))
             (lambda (env) (set-variables-value! var val (enclosing-environment env)))
             (lambda () (error "Unbound variable -- set-variable-value!" var)))))

(define (define-variable! var val env)
  (tranverse var
             env
             (lambda (vals) (set-car! vals val))
             (lambda (env) (add-binding-to-frame! var val (first-frame env)))
             (lambda () (error "Empty environment -- define-variable!"))))
```

meteorgan

```
; this solution is based on exercise 4.11, that's to say i used different frame.
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too few arguments supplied" vars vals)
          (error "Too many arguments supplied" vars vals)))))

;; look up a variable in a frame
(define (lookup-binding-in-frame var frame)
  (cond ((null? frame) (cons false '()))
        ((eq? (car (car frame)) var)
         (cons true (cdr (car frame))))
        (else (lookup-binding-in-frame var (cdr frame)))))

;; in frame, set var to val
(define (set-binding-in-frame var val frame)
  (cond ((null? frame) false)
        ((eq? (car (car frame)) var)
         (set-cdr! (car frame) val)
         true)
        (else (set-binding-in-frame var val (cdr frame)))))

(define (lookup-variable-value var env)
  (if (eq? env the-empty-environment)
```

```

(error "Unbound variable" var))
(let ((result (lookup-binding-in-frame var (first-frame env))))
  (if (car result)
      (cdr result)
      (lookup-variable-value var (enclosing-environment
env)))))

(define (set-variable-value! var val env)
  (if (eq? env the-empty-environment)
      (error "Unbound variable -- SET" var)
      (if (set-binding-in-frame var val (first-frame env))
          true
          (set-variable-value! var val (enclosing-environment env)))))

(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (if (set-binding-in-frame var val frame)
        true
        (set-car! env (cons (cons var val) frame)))))


```

SophiaG

```

(define (env-loop env base match)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars)
              base)
            ((eq? var (car vars))
              match)
            (else (scan (cdr vars) (cdr vals))))))
    (scan (frame-variables frame)
          (frame-values frame)))

(define (lookup-variable-value var env)
  (env-loop env
            (env-loop (enclosing-environment env))
            (car vals)))

(define (set-variable-value! var val env)
  (env-loop env
            (env-loop (enclosing-environment env))
            (set-car! vals val)))

(define (define-variable! var val env)
  (env-loop env
            (add-binding-to-frame! var val frame)
            (set-car! vals val)))


```

mazj

something need be change in SophiaG's answer, the "next" and "match" need be a lambda but not a exp.

```

(define (env-loop env next match)
  (define (scan vars vals)
    (cond ((null? vars) (next env))
          ((eq? var (car vars)) (match vals))
          (else (scan (cdr vars) (cdr vals))))))
  (if (eq? env the-empty-environment)
      (error "Unbound variable" var)
      (let ((frame (first-frame env)))
        (scan (frame-variables frame)
              (frame-values frame)))))

(define (lookup-variable-value var env)
  (env-loop env
            (lambda (env) (env-loop (enclosing-environment env)))
            car))

(define (set-variable-value! var val env)
  (env-loop env
            (lambda (env) (env-loop (enclosing-environment env)))
            (lambda (vals) (set-car! vals val)))))

(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (env-loop env
              (lambda (env) (add-binding-to-frame!
                            var val (first-frame env)))
              (lambda (vals) (set-car! vals val))))))


```

Ada

```

;; general procedure
(define (env-loop match-proc end-frame end-env env)
  (define (scan vars vals current-frame current-env)
    (cond ((null? vars)
           (end-frame current-frame current-env))
          ((eq? var (car vars))
           (match-proc vars vals current-frame current-env))
          (else
            (scan (cdr vars) (cdr vals) current-frame current-env))))
  (if (eq? env the-empty-environment)
      (end-env)
      (let ((frame (first-frame env)))
        (scan (frame-variables frame)
              (frame-values frame)
              frame env)))))

;; lookup-variable-value
(define (lookup-variable-value var env)
  (define (match-proc vars vals cur-frame cur-env) (car vals))
  (define (end-env) (error "Unbound variable" var))
  (define (end-frame cur-frame cur-env) ;!!!
    (env-loop match-proc end-frame end-env (enclosing-environment cur-env)))
  (env-loop match-proc end-frame end-env env))

;; set-variable-value!
(define (set-variable-value! var val env)
  (define (match-proc vars vals cur-frame cur-env) (set-car! vals val))
  (define (end-env) (error "Unbound variable" var))
  (define (end-frame cur-frame cur-env) ;!!!
    (env-loop match-proc end-frame end-env (enclosing-environment cur-env)))
  (env-loop match-proc end-frame end-env env))

;; define-variable!
(define (define-variable! var val env)
  (define (match-proc vars vals cur-frame cur-env) (set-car! vals val))
  (define (end-env) (error "Unbound variable" var))
  (define (end-frame cur-frame cur-env) ;!!!
    (add-binding-to-frame! var val cur-frame))
  (env-loop match-proc end-frame end-env env))

```

CrazyAlvaro

I don't think SophiaG's solution is correct, since you can't just pass 'text' to be evaluated later

```

(define (scan-frame var frame no-vars-callback found-callback)
  (define (scan variables values)
    (cond ((null? variables)
           (no-vars-callback))
          ((eq? var (car variables))
           (found-callback variables values))
          (else (scan (cdr variables) (cdr values)))))
  (scan (frame-variables frame) (frame-values frame)))

(define (search-env var env success-callback)
  (define no-found-callback
    ; search the next environment
    (search-env var (enclosing-environment env) success-callback))
  (if (eq? env the-empty-environment)
      (error "Unbound variable" var)
      (scan-frame var (first-frame env) no-found-callback success-callback)))

(define (lookup-variable-value var env)
  (search-env var env (lambda (vars vals) (car vals))))

(define (set-variable-value! var val env)
  (search-env var env (lambda (vars vals) (set-car! vals val))))

(define (define-variable! var val env)
  (scan-frame
    var
    (first-frame env)
    (lambda () (add-binding-to-frame! var val frame)
    (lambda (vars vals) (set-car! vals val)))))


```

poly

...

```
(define (env-loop var-not-in-frame proc env var)
  (define (scan frame)
    (define (iter vars vals)
      (cond ((null? vars) (var-not-in-frame))
            ((eq? var (car vars)) (proc vals))
            (else
              (iter (cdr vars) (cdr vals)))))
      (iter (frame-variables frame) (frame-values frame)))

  (if (eq? env the-empty-environment)
      (error "Unbound variable" var)
      (let ((frame (first-frame env)))
        (or (scan frame) ; the iteration won't be activated if the value of
            ; (scan frame) isn't false, which means the action
            ; here is depended on value of (var-not-in-frame)
        (env-loop var-not-in-frame proc (enclosing-environment env) var)))))

(define (lookup-variable-value var env)
  (define (var-not-in-frame) false)
  (env-loop var-not-in-frame car env var))

(define (set-variable-value! var val env)
  (define (var-not-in-frame) false)
  (env-loop var-not-in-frame (lambda (x) (set-car! x val)) env var))

(define (define-variable! var val env)
  (define (var-not-in-frame)
    (add-binding-to-frame! var val (first-frame env)))
  (env-loop var-not-in-frame (lambda (x) (set-car! x val)) env var))
```

I just found out there is bug: if var is bound to "false" then the scan will return false and env-loop will keep on iteration until it hits an error.

new one:

```
(define (env-loop env var var-not-in-frame proc)
  (define (scan vars vals)
    (cond ((null? vars) (var-not-in-frame env))
          ((eq? var (car vars)) (proc vals))
          (else
            (scan (cdr vars) (cdr vals)))))
  (if (eq? env the-empty-environment)
      (error "Unbound variable" var)
      (let ((frame (first-frame env)))
        (scan (frame-variables frame)
              (frame-values frame)))))

(define (lookup-variable-value var env)
  (define (var-not-in-frame env)
    (lookup-variable-value var (enclosing-environment env)))
  (env-loop env var var-not-in-frame car))

(define (set-val! val)
  (lambda (vals) (set-car! vals val)))

(define (set-variable-value! var val env)
  (define (var-not-in-frame env)
    (set-variable-value! var val (enclosing-environment env)))
  (env-loop env var var-not-in-frame (set-val! val)))

(define (define-variable! var val env)
  (define (var-not-in-frame env)
    (add-binding-to-frame! var val (first-frame env)))
  (env-loop env var var-not-in-frame (set-val! val)))
```

aos

I think the easiest way to solve this is to just pass a message as to which method called env-loop. For example:

```
(define (env-loop var val env action)
  (define (scan vars vals)
    (cond ((and (eq? env the-empty-environment)
                (or (eq? action 'lookup-var)
                    (eq? action 'set-var!)))
           (error "Undefined variable" var))
          ((null? vars)
```

```

(if (or (eq? action 'lookup-var)
         (eq? action 'set-var!))
    (env-loop
      var
      val
      (enclosing-environment env)
      action)
    (add-binding-to-frame!
      var val (first-frame env))))
((eq? var (car vars))
  (if (or (eq? action 'define-var)
          (eq? action 'set-var!))
      (set-car! vals val)
      (car vals)))
  (else (scan (cdr vars)
              (cdr vals)))))
(let ((frame (first-frame env)))
  (scan (frame-variables frame)
        (frame-values frame)))

(define (lookup-variable-value var env)
  (env-loop var '() env 'lookup-var))

(define (set-variable-value! var val env)
  (env-loop var val env 'set-var!))

(define (define-variable! var val env)
  (env-loop var val env 'define-var))

```

Here we can just check against the `action` parameter to determine what action to take! The important information that does not change is captured in the `env-loop` environment (such as `var`, `val`, `env`, and `action`) and `scan` does not have to worry about it.

revc

All procedures try to find the specified variable in the environment, if any, the procedure will perform the corresponding actions; if not, it will take other actions.

```

#lang racket
(require compatibility/mlist)

;; representing frames
(define (make-frame variables values)
  (mcons variables values))

(define (frame-variables frame) (mcadr frame))
(define (frame-values frame) (mcdr frame))

(define (add-binding-to-frame! var val frame)
  (set-mcar! frame (mcons var (mcadr frame)))
  (set-mcdr! frame (mcons val (mcdr frame))))


;; representing environments (a list of frames)
(define (enclosing-environment env) (mcdr env))
(define (first-frame env) (mcadr env))
(define the-empty-environment empty)

;; operations on environments

(define (find-var-and-do var env unfound found)
  (define (scan vars vals)
    (cond [(null? vars) (unfound)]
          [(eq? var (mcadr vars)) (found vals)]
          [else (scan (mcddr vars) (mcdr vals))]))
  (if (eq? env the-empty-environment)
      (error "Unbound variable" var)
      (let ([frame (first-frame env)])
        (scan (frame-variables frame)
              (frame-values frame)))))

;; return a value
(define (lookup-variable-value var env)
  (find-var-and-do var env
    (λ () (lookup-variable-value var (enclosing-environment env)))
    (λ (vals) (mcadr vals))))


;; add a new binding
(define (define-variable! var val env)
  (find-var-and-do var env
    (λ () (add-binding-to-frame! var val (first-frame env)))
    (λ (vals) (set-mcar! vals val))))

```

```


;;; change an existed binding
(define (set-variable-value! var val env)
  (find-var-and-do var env
    (λ () (set-variable-value! var val (enclosing-environment env)))
    (λ (vals) (set-mcar! vals val)))))



;;; return new environment
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (mcons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))
```

o3o3o There are two parts: `traverse-env` and `traverse-frame`, each of which just do simple thing:

1. if find or match by `var`, return matched `(var, val)`;
2. if-not return `NOT-MATCH`.

And we need some action on the upper function according the returned result of `traverse-*` function.

I think using callback is not good way, because the callbacks need some args which should not be fixed in some conditions.

I also tried to use call-back with call-back text which is going to be called by `eval`. There are two reason not to use call-back:

1. `eval` need a `env` argument;
2. The invoker writing call-back function has to jump into the `traverse-*` to see what to invoke.

```


(define (traverse-env env var match-action)
  (define (env-loop-iter env)
    (if (eq? env the-empty-environment)
        "NOT-MATCH"
        (let* ((frame (first-frame env)))
          (res (traverse-frame frame var))
          (if (eq? res "NOT-MATCH")
              (env-loop-iter (enclosing-environment env))
              (list frame res)))); ; return (frame (vars vals) if matched
  (env-loop-iter env))

(define (traverse-frame frame var match-action not-match-action)
  (define (scan-iner vars vals)
    (cond ((null? vars)
           "NOT-MATCH"
           ((eq? var (car vars))
            (list vals vals)) ; return (vars vals) if matched
           (else
             (scan-iner (cdr vars) (cdr vals))))))
    (scan-iner (frame-variables frame)
               (frame-values frame)))

  (define (lookup-variable-value var env)
    (let (res (traverse-env env var))
      (if (eq? res "NOT-MATCH")
          (error "Unbound variable" var)
          (caaddr res))); ; (frame (vars vals)) -> val

  (define (set-variable-value! var val env)
    (let (res (traverse-env env var))
      (if (eq? res "NOT-MATCH")
          (error "Unbound variable -- SET!" var)
          (set-car! ((cadadr res) val)))); ; (frame (vars vals)) -> vals

  (define (define-variable! var val env)
    (let* ((frame (first-frame env)))
      (res (traverse-frame frame))
      (if (eq? "NOT-MATCH")
          (add-binding-to-frame! var val frame)
          (set-car! ((caddr res) val)))); ; (vars vals) -> vals
```

Sphinxsky

I think exercise 4-11 and exercise 4-12 should be done together!

```

; The most important abstraction!!!
(define (scan items is-over? is-it? get-now get-other)
  (if (is-over? items)
      #f
      (let ((now (get-now items)))
        (if (is-it? now)
            now
            (scan (get-other items) is-over? is-it? get-now get-other)))))

; define binding
(define (make-binding var val)
  (cons var val))
(define (binding-var binding)
  (car binding))
(define (binding-val binding)
  (cdr binding))
(define (set-binding! binding new-val)
  (set-cdr! binding new-val))

; define frame (exercise 4-11)
(define (make-frame variables values-)

  (define (lookup var bindings)
    (define (is-it? now)
      (eq? var (binding-var now)))
    (scan bindings null? is-it? car cdr))

  (let ((bindings (map make-binding variables values-)))
    (lambda (msg)
      (cond ((eq? msg 'add)
             (lambda (binding)
               (set! bindings (cons binding bindings))))
            ((eq? msg 'lookup)
             (lambda (var)
               (lookup var bindings)))
            ((eq? msg 'vars)
             (map binding-var bindings))
            ((eq? msg 'vals)
             (map binding-val bindings))
            (else (error "Unknown operation -- FRAME" msg)))))

  (define (frame-variables frame)
    (frame 'vars))
  (define (frame-values frame)
    (frame 'vals))
  (define (add-binding-to-frame! var val frame)
    ((frame 'add) (make-binding var val)))
  (define (lookup-binding-to-frame var frame)
    ((frame 'lookup) var))

; the code in book
(define (enclosing-environment env)
  (cdr env))
(define (first-frame env)
  (car env))
(define the-empty-environment '())

; define environment
(define (is-empty-environment? env)
  (eq? env the-empty-environment))

(define (lookup-binding-to-environment var env)
  (define (get-now env)
    (lookup-binding-to-frame var (first-frame env)))
  (scan
    env
    is-empty-environment?
    (lambda (now) now)
    get-now
    enclosing-environment))

; The rewriting process makes the scope of variables more friendly
(define (extend-environment vars vals base-env)
  (let ((vars-len (length vars))
        (vals-len (length vals)))
    (if (= vars-len vals-len)
        (if (is-empty-environment? base-env)
            (cons (make-frame vars vals) base-env)
            (let ((ff (first-frame base-env))
                  (ee (enclosing-environment base-env)))
              (set-cdr! ff ee)
              ee))
        (cons (make-frame vars (cdr vals)) (cdr base-env))))
```

```

(set-car! base-env (make-frame vars vals))
(set-cdr! base-env (cons ff ee))
base-env)
(if (< vars-len vals-len)
(error "Too many arguments supplied" vars vals)
(error "Too few arguments supplied" vars vals)))))

; ===== exercise 4-12 =====
(define (lookup-variable-value var env)
(let ((binding (lookup-binding-to-environment var env)))
(if binding
(binding-val binding)
(error "Unbound variable" var)))))

(define (set-variable-value! var val env)
(let ((binding (lookup-binding-to-environment var env)))
(if binding
(set-binding! binding val)
(error "Unbound variable -- SET!" var)))))

(define (define-variable! var val env)
(let* ((frame (first-frame env))
(binding (lookup-binding-to-frame var frame)))
(if binding
(set-binding! binding val)
(add-binding-to-frame! var val frame))))
```

master

To define variables we only need to search the current frame, for the other two procedures we need to go through all frames and recurse into each one separately. Seems like a good opportunity for abstraction! `search-frame` searches the bindings in the current frame, and `search-environment` searches through the environment by calling `search-frame` for each frame in sequence.

```

(define (search-environment var env match nomatch error-message)
(define (rec env)
(if (eq? env the-empty-environment)
(error error-message var)
(let ((frame (first-frame env)))
(search-frame var frame match nomatch)))
(rec env)))

(define (search-frame var frame match nomatch)
(define (rec bindings)
(cond ((null? bindings) (nomatch))
((let ((res (assoc var bindings)))
(match res)))
(else (rec (cdr bindings)))))
(rec (frame-bindings frame)))

(define (lookup-variable-value var env)
(search-environment var
env
(lambda (res) (and res (cdr res)))
(lambda () (lookup-variable-value var (enclosing-environment env)))
"Unbound variable"))

(define (set-variable-value! var val env)
(search-environment var
env
(lambda (res) (and res (set-cdr! res val)))
(lambda () (set-variable-value! var val (enclosing-environment
env))))
"Unbound variable: SET!")

(define (define-variable! var val env)
(let ((frame (first-frame env)))
(search-frame var
frame
(lambda (res) (and res (set-cdr! res val)))
(lambda () (add-binding-to-frame! var val frame))))
```

SteeleDynamics

(master)'s solution above is correct and is (almost???) Continuation-Passing Style (CPS). Correct me if I am wrong. I think (???) my solution is correctly implemented in CPS, where `sc` is the success continuation and `fc` is the failure continuation.

```

; frame-lookup-cps procedure
(define (frame-lookup-cps var vars vals sc fc) ;!
```

```

(cond ((null? vars) (fc))
      ((eq? var (car vars)) (sc vals))
      (else (frame-lookup-cps var (cdr vars) (cdr vals) sc fc)))))

; env-lookup-cps procedure
(define (env-lookup-cps var env sc fc) ;!
  (if (eq? env the-empty-environment)
      (fc)
      (let ((frame (first-frame env)))
        (let ((vars (frame-variables frame))
              (vals (frame-values frame))
              (enc (enclosing-environment env)))
          (let ((fc (lambda () (env-lookup-cps var enc sc fc))))
            (frame-lookup-cps var vars vals sc fc)))))

; lookup-variable-value procedure
(define (lookup-variable-value var env) ;!
  (env-lookup-cps
   var
   env
   (lambda (vals) (car vals))
   (lambda () (error "Unbound variable" var)))))

; set-variable-value! mutator procedure
(define (set-variable-value! var val env) ;!
  (env-lookup-cps
   var
   env
   (lambda (vals) (set-car! vals val))
   (lambda () (error "Unbound variable -- SET!" var)))))

; define-variable! mutator procedure
(define (define-variable! var val env) ;!
  (let ((frame (first-frame env)))
    (let ((vars (frame-variables frame))
          (vals (frame-values frame)))
      (frame-lookup-cps
       var
       vars
       vals
       (lambda (vals) (set-car! vals val))
       (lambda () (add-binding-to-frame! var val frame)))))))

```

# sicp-ex-4.13

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.12) | Index | Next exercise (4.14) >>

meteorgan

```
; unbound variable in current frame
(define (unbound? expr) (tagged-list? expr 'unbound))
(define (unbind-variable expr env) (make-unbound (cadr expr) env))
(define (make-unbound variable env)
  (let ((vars (frame-variables (first-frame env)))
        (vals (frame-values (first-frame env))))
    (define (unbound vars vals new-vars new-vals)
      (cond ((null? vars)
             (error "variable is not in the environment -- MAKE-
UNBOUND"
                   variable))
            ((eq? (car vars) variable)
             (set-car! env
                       (cons (append new-vars (cdr vars))
                             (append new-vals (cdr vals))))))
            (else (unbound (cdr vars) (cdr vals)
                           (cons (car vars) new-vals)
                           (cons (car vals) new-vals))))))
    (unbound vars vals '() '())))
  
;; add this in eval
(unbound? expr) (unbind-variable expr env))
```

wing

```
;there's no need to unbind bindings in the enclosing environments, and no
need to send an error message(just like "define"), a simpler version is:
(define (make-unbound! var env)
  (let* ((frame (first-frame env))
        (vars (frame-variables frame))
        (vals (frame-values frame)))
    (define (scan pre-vals pre-vars vars vals)
      (if (not (null? vars))
          (if (eq? var (car vars))
              (begin (set-cdr! pre-vals (cdr vars))
                     (set-cdr! pre-vals (cdr vals)))
                  (scan vars vals (cdr vars) (cdr vals))))
          (if (not (null? vars))
              (if (eq? var (car vars))
                  (begin (set-car! frame (cdr vars))
                         (set-cdr! frame (cdr vals)))
                      (scan vars vals (cdr vars) (cdr vals)))))))
```

mazj

wing's answer is more 4 times quick than my:

```
(define (make-unbound! var env) ;4*linear
  (let ((frame (first-frame env)))
    (define pairs
      (filter (lambda (pair) (not (eq? (car pair) var-target)))
              (map cons (car frame) (cdr frame)))))
    (set-car! frame (map car pairs))
    (set-cdr! frame (map cdr pairs))))
```

master

This isn't that elegant. The problem is that the way I wrote `search-frame` it returns the exact `cons` cell where the match occurred, so there really isn't any way to access the surrounding frame. Could easily be picked up by a garbage collector though.

```
(define (make-unbound! var env)
  (let ((frame (first-frame env)))
    (search-frame var
      frame
      (lambda (res) (and res (set-car! res '()) (set-cdr! res '())))
      (lambda () '()))))
```

Shade

I believe 'make-unbound!' should construct a list for the evaluator to check with 'unbound?', and only afterwards it calls the unbinding procedure:

```
(define (make-unbound! var)
  (list 'unbound! var))
(define (unbound? exp) (tagged-list? exp 'unbound!))
(define (unbound!-var exp)
  (cadr exp))
(define (unbound! var env)
  (define (remove-binding var bindings)
    (cond ((null? bindings) '())
          ((eq? var (caar bindings)) (cdr bindings))
          (else (cons (car bindings)
                       (remove-binding var (cdr bindings)))))))
  (let ((frame (first-frame env)))
    (set-cdr! frame (remove-binding var (frame-bindings frame))))))
```

pvk

Here's both options:

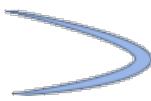
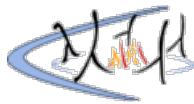
```
(define (make-unbound-in-frame! var frame)
  (define (scan bindings-before bindings-after)
    (cond ((null? bindings-after)
           bindings-before)
          ((eq? var (caar bindings-after))
           (append bindings-before (cdr bindings-after)))
          (else
            (scan (append bindings-before (list (car bindings-after)))
                  (cdr bindings-after))))
    (set-cdr! frame (scan '() (cdr frame))))
  (define (make-unbound-local! var env)
    (make-unbound-in-frame! (var (first-frame env))))
  (define (make-unbound! var env)
    (if (eq? env the-empty-environment)
        'done
        (begin (make-unbound-local! var env)
               (make-unbound! var (enclosing-environment env))))))
```

My frames are in format `(*frame* (var1 val1) (var2 val2) ...)` (as in [sicp-ex-4.11](#)). I didn't find the loops I wrote for [sicp-ex-4.12](#) useful, as my `scan-frame` could only do anything to the final segment of the frame-bindings starting with the variable it was looking for.

As for the question the book asked, I found myself strongly preferring to unbind variables globally, for the following reasons:

1. `(make-unbound! var env)` guarantees to the programmer that a subsequent lookup of `var` in `env` will return an "Unbound variable" error. `(make-unbound-local! var env)` can provide no such guarantee (and notice that we've so far given the programmer no tools for looking up a variable in a `frame`).
2. If we only have `make-unbound-local!`, we can't unbind variables from inside procedure calls. The major downside of global `make-unbound!` is that a programmer might accidentally unbind a variable from a package they had installed. In my opinion, this is a problem that should be dealt with in the package loading system.

# sicp-ex-4.14



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.13) | Index | Next exercise (4.15) >>

meteorgan

```
;; Parse error: Spurious closing paren found
```

Eva's map can work, because he implemented it. But when installing map in the eval as a primitive procedure, there is something wrong. for example:  
when eval expression '(map + (1 2) (3 4))', primitive procedure + is interpreted as '(application + env)', so the expression is (apply map (list 'application + env) (list 1 2) (list 3 4))), it doesn't work.

codybartfast

I get the following error with Louis's approach:

```
application: not a procedure;  
expected a procedure that can be applied to arguments  
given: #0=(mcons 'procedure (mcons (mcons 'x '()) (mcons (mcons (mcons  
'* (mcons 'x (mcons 'x (mcons 'x '())))) '()) (mcons (mcons (mcons  
'*frame* (mcons (mcons 'cube #0#) (mcons (mcons 'false #f) (mcons  
(mcons 'true #t) (mcons (mcons 'map #<procedure:mm...  
arguments....:
```

I believe the problem is that Louis is 'crossing the streams'. He is passing one of 'our' procedures to a primitive procedure.

We have a procedure, cube, that is designed to be applied in the implementation that we are constructing. But here we are passing cube (i.e. the procedure object referenced by the symbol 'cube) to the underlying map. So it will be the underlying implementation (e.g. Racket, Guile, Chicken, ...) that will apply map and, in turn, attempt to apply our cube procedure. This cannot work if our implementation of procedures, environments, etcetera is different from the ones used by the underlying implementation.

(If key parts of the underlying implementation were identical to our implementation I can imagine that it might work, but clearly we should never rely on that.)

revc

The procedures of metacircular evaluator(which are represented as lists consisting of a function and an environment) are not same as the procedures of scheme, so we need meta-apply(the apply of metacircular) to tackle that problem.

```
(define meta-map  
  (lambda (f l)  
    (if (null? l)  
        '()  
        (cons (meta-apply f `((, (car l)))) (meta-map f (cdr l))))))
```

Shi-Chao

```
(define primitive-procedures  
  (list 'car car)  
        (list 'cdr cdr)  
        (list 'cons cons)  
        (list 'null? null?)  
        (list 'map map)))  
  
;;; test
```

```
car
(map car '((1 a) (2 b) (3 c)))

;; M-Eval input:
car
;; M-Eval value:
(primitive #[compiled-procedure 12 (list #x1) #x1a #x10305319a])

;; M-Eval input:
(map car '((1 a) (2 b) (3 c)))
;The object (primitive #[compiled-procedure 12 ("list" #x1) #x1a #x10305319a]) is not
applicable.
```

Error message above said (**primitive car**) is not applicable. The point is that within the Scheme's implementation of **map**, **car** will be applied to the list, but the **car** is interpreted in the form ('**primitive car**) in our evaluator which is not recognized in Scheme's evaluator.

yd

The point here is, by the definition of <eval>, evaluation of <map> will lead to eval of the first operand, <+>. Interpreter does it by searching the initial env which we setup at very first, and find that <+> is bind with the list ('primitive +). And since map is primitive in the implemented lisp, procedure <apply> will pass the whole list to original "map" as the first operand, but it is a procedure which implemented in our lisp, also not a legal form that can be recognized by underlying lisp system.

# sicp-ex-4.15

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.14) | Index | Next exercise (4.16) >>

meteorgan

assuming procedure try can halt, when execute (try try), (halt? try try) will be true, but the procedure will run forever. **if** procedure try can't halt. (halt? try try) will be false, but the procedure will halt.

lain

This is proof by contradiction, where we assume the opposite of what we want to prove and show that it is logically inconsistent.

If we call (try try) then we get the following (with try substituted in for p)

```
(define (try try)
  (if (halts? try try)
      (run-forever)
      'halted))
```

We observe that if (halts? try try) is True then (try try) will run forever, which is a contradiction. The converse, if (halts? try try) if False then (try try) returns 'halted, could also be used to reach a contradiction.

# sicp-ex-4.16

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.15) | Index | Next exercise (4.17) >>

woofy

```
; a
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars) (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (if (eq? '*unassigned* (car vals))
                 (error "Variable Unassigned -- LOOKUP-VARIABLE-VALUE" var)
                 (car vals)))
            (else (scan (cdr vars) (cdr vals))))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))

; b
(define (make-let bingings body)
  (cons 'let (cons bingings body)))

(define (make-assignment var exp)
  (list 'set! var exp))

(define (scan-out-defines body)

  (define (collect seq defs exps)
    (if (null? seq)
        (cons defs exps)
        (if (definition? (car seq))
            (collect (cdr seq) (cons (car seq) defs) exps)
            (collect (cdr seq) defs (cons (car seq) exps)))))

  (let ((pair (collect body '() '())))
    (let ((defs (car pair)) (exp (cdr pair)))
      (make-let (map (lambda (def)
                      (list (definition-variable def)
                            '*unassigned*))
                     defs)
                (append
                  (map (lambda (def)
                          (make-assignment (definition-variable def)
                                          (definition-value def)))
                     defs)
                  exps)))))

; c
; make-procedure is better because we can easily explore other transformations
; along with the fact of repeated calculation everytime when procedure-body is accessed

(define (make-procedure parameters body env)
  (list 'procedure parameters (scan-out-defines body) env))

or

(define (make-procedure-with-transformation transformation)
  (define (make-procedure parameters body env)
    (list 'procedure parameters (transformation body) env))
  make-procedure)
```

meteorgan

```
; ; a, change look-up-variable-value
```

```

(define (lookup-variable-value var env)
  (define (env-lookup env)
    (define (scan vars vals)
      (cond ((null? vars) (env-lookup (enclosing-environment env)))
            ((eq? var (car vars))
             (if (eq? (car vals) '*unassigned*)
                 (error "variable is unassigned" var)
                 (car vals)))
            (else (scan (cdr vars) (cdr vals))))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))

  (env-lookup env))

;; b
(define (scan-out-defines body)
  (define (name-unassigned defines)
    (map (lambda (x) (list (definition-variable x) '*unassigned*)) defines))
  (define (set-values defines)
    (map (lambda (x)
            (list 'set! (definition-variable x) (definition-value x)))
         defines))
  (define (defines->let exprs defines not-defines)
    (cond ((null? exprs)
           (if (null? defines)
               body
               (list (list 'let (name-unassigned defines)
                           (make-begin (append
                                         (set-values defines)
                                         (reverse not-defines)))))))
          ((definition? (car exprs))
           (defines->let (cdr exprs) (cons (car exprs) defines) not-defines))
          (else (defines->let (cdr exprs) defines (cons (car exprs) not-defines)))))

  (defines->let body '() '()))

;; c
install scan-out-defines into make-procedure. otherwise, when we call procedure-body,
procedure scan-out-defines will be called.

```

fubupc

Why move all set! before any other expressions? The book seems not require that.

atupal

```

; Start Exercise 4.16
;a
(define (lookup-variable-value-4.16a var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars)) (car vals))
            (else (scan (cdr vars) (cdr vals)))))

    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))

  (let ((value (env-loop env)))
    (if (eq? value '*unassigned*)
        (error "Unassigned variable: *unassigned*")
        value)))

(define lookup-variable-value lookup-variable-value-4.16a)
;b
(define (split-body-out-defines body)
  (if (null? body)
      (let ((defines '())
            (others '()))
        (cons defines others))
      (let ((exp (car body))
            (rest (split-body-out-defines (cdr body))))
        (if (definition? exp)
            (cons (cons exp (car rest)) (cdr rest))
            (cons (car rest) (cons exp (cdr rest)))))))

```

```

(define (make-let varvals body)
  (list 'let varvals body))
(define (defines->let-defines-body defines)
  (if (null? defines)
      (let ((let-defines '()))
        (let-body '())
        (cons let-defines let-body))
      (let* ((rest-let-defines-body (defines->let-defines-body (cdr defines)))
             (rest-defines (car rest-let-defines-body))
             (rest-body (cdr rest-let-defines-body))
             (name (definition-variable (car defines)))
             (value (definition-value (car defines))))
        (current-define (list name ''unassigned))
        (current-body (list 'set! name value)))
        (cons (cons current-define rest-defines)
              (cons current-body rest-body)))))

(define (scan-out-defines procedure-body)
  (let* ((splited-body (split-body-out-defines procedure-body))
         (defines (car splited-body))
         (others (cdr splited-body))
         (let-defines-body (defines->let-defines-body defines)))
    (list (append (list 'let
                         (car let-defines-body))
                  (append (cdr let-defines-body)
                          others))))))

;c
(define (contain-defines exps)
  (if (null? exps)
      false
      (or (if (definition? (car exps))
              true
              false)
          (contain-defines (cdr exps)))))

(define (make-procedure-ex4.16 parameters body env)
  (if (contain-defines body)
      (list 'procedure parameters (scan-out-defines body) env)
      (list 'procedure parameters body env)))
(define make-procedure make-procedure-ex4.16)

```

wing

```

;;note that if there aren't any definitions at all,you have to keep the
original form of procedure body,otherwise an infinite recursion will be caused.
;;consider the evaluating of (let () 5),which is very interesting
(define (make-let bindings body)
  (cons 'let (cons bindings body)))
(define (scan-out-defines body)
  (define (append x y)
    (if (null? x) y (cons (car x) (append (cdr x) y))))
  (let* ((definitions
          (filter (lambda (x)
                    (and (pair? x) (eq? (car x) 'define))) body))
         (non-definitions
          (filter (lambda (x)
                    (or (not (pair? x))
                        (not (eq? (car x) 'define)))) body)))
    (let-vars (map definition-variable definitions))
    (letVals (map definition-value definitions))
    (let-bindings
      (map (lambda (x) (list x ''unassigned)) let-vars))
    (assignments
      (map (lambda (x y) (list 'set! x y)) let-vars letVals)))
  (if (null? let-bindings)
      body
      (list (make-let let-bindings (append assignments non-definitions)))))))

```

dzy

```

(define (make-let vars body)
  (if (null? vars)
      body
      (let ((inlet (map (lambda (x) (list (car x) ''unassigned)) vars))
            (sets (map (lambda (x) (list 'set! (car x) (cdr x))) vars)))
        (list (append (list 'let inlet) (append sets body)))))

(define (scan-out-defines body)
  (define (iter vars rest body)
    (cond ((null? body)
           (make-let vars rest))
          ((definition? (car body))
           (iter (append vars (list (cons (definition-variable (car body)

```

```

        (definition-value (car body))))))
      rest (cdr body)))
(else
  (iter vars
    (append rest (list (car body)))
    (cdr body))))))
(iter '() '() body))

```

be careful if there's no definition. (let () <body>) will cause infinitely recurse.

master

I don't know where you all got the impression that (let () <body>) causes an infinite recursion... It's ugly but it definitely does *not* do any such thing. I tried it on four different Scheme implementations and all of them evaluate such an expression correctly. In fact (let () <body>) is exactly equivalent to ((lambda () <body>)), which is absolutely fine. Also seeing as it's just an internal representation it doesn't really matter how ugly it is. It is still valid to want to reserve the original procedure, just not necessary. Please correct me if I'm wrong.

Here's my solution:

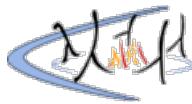
```

;; a
(define (lookup-variable-value var env)
  (search-environment var
    env
    (lambda (res) (and res (if (eq? res '*unassigned*)
                                (error "Unassigned variable" var)
                                (cdr res))))
    (lambda () (lookup-variable-value var (enclosing-environment env)))
    "Unbound variable"))

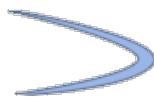
;; b
(define (scan-out-defines proc-body)
  (define (iter exps vars)
    (if (null? exps)
        vars
        (let ((this-exp (car exps)))
          (if (definition? this-exp)
              (iter (cdr exps) (cons (cons (definition-variable this-exp)
                                            (list (definition-value this-exp)))
                                      vars))
              (iter (cdr exps) vars))))
    (let* ((body (lambda-body proc-body))
           (vars (reverse (iter body '()))))
      (make-lambda (lambda-parameters proc-body)
        (list (make-let (map (lambda (x) (list (car x) '*unassigned*)) vars)
                      (append (map (lambda (x) (list 'set! (car x) (cadr x))) vars)
                            (filter (lambda (x) (not (definition? x)))) body)))))))

;; c
(define (make-procedure parameters body env)
  (list 'procedure parameters (scan-out-defines body) env))

```



# sicp-ex-4.17



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.16) | Index | Next exercise (4.18) >>

meteorgan

**;; a**  
because `let` expression will be substituted as `lambda` expression, every `lambda` expression will extend the environment.

**;; b**  
the new environment is confined to the `let` expression, it doesn't change the outer environment.

**;; c**  
Do away with the `let` - All the (`define` var '\*unassigned\*) which was there in the `let` statements should instead be moved on top of the body. The `set!` statements should simply replace the earlier defines.

Note that you cannot simply move all defines to the start of the body, because that might allow the body to access variables **not** yet defined in the original code.

pvk

Sequential definition:

```
((lambda <vars>
  (define u <e1>)
  (define v <e2>)
  <e3>) <inputs>)
```

will evaluate `<e3>` in the environment:

```
GLOBAL
-----
FRAME1 (<e3> evaluated here)
-----
<vars>: <inputs>
u: <e1>
v: <e2>
```

(where each frame is a child of the one above it.)

Simultaneous definition:

```
((lambda <vars> (let ((u '*unassigned*)
                         (v '*unassigned*))
  (set! u <e1>)
  (set! v <e2>)
  <e3>)) <inputs>)
```

expands to

```
((lambda <vars> (lambda (u v)
  (set! u <e1>)
  (set! v <e2>)
  <e3>)
  '*unassigned* '*unassigned*)
 <inputs>)
```

and since there are two lambdas, there are two frames:

```
GLOBAL
-----
FRAME1
-----
<vars>: <inputs>
```

```

FRAME2 (<e3> evaluated here)
-----
u: <e1>
v: <e2>

```

It's pretty clear that evaluating an expression in either environment will give the same values for all variables, and thus the same result.

In addition to @meteorgan's idea of replacing `(let ((u '*unassigned*) (v '*unassigned*)) ...)` with `(define u '*unassigned*) (define v '*unassigned*) ...`, one can imagine a solution that would essentially translate the whole expression to

```

((lambda (<vars> u v)
  (set! u <e1>)
  (set! v <e2>)
  <e3>)
 <inputs> '*unassigned* '*unassigned*)

```

Doing this would require modifying how procedures are applied to inputs, i.e., replace

```

(eval-sequence (procedure-body procedure)
               (extend-environment (procedure-parameters procedure)
                                   arguments
                                   (procedure-environment procedure)))

```

in the definition of `apply` with

```

(eval-sequence (procedure-body procedure)
               (extend-environment (append (procedure-parameters procedure)
                                         (internally-defined-vars procedure))
                                   (append arguments (unassigneds procedure))
                                   (procedure-environment procedure)))

```

with appropriate definitions of the undefined terms, and modifications to the internal representation of procedures. I think this is all pretty academic, so excuse me if I leave out the details.

# sicp-ex-4.18

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.17) | Index | Next exercise (4.19) >>

meteorgan

this won't work. because, in `(let ((a <e1>) (b <e2>)))`, when compute e2, it depends y, but we only have a `not` y. For the same reason, the solution in text will work.

uuu

I think both method will work, because `(eval "dy")` is 'delayed'.

LuckyKoala

But the expression "`(stream-map f y)`" defined in dy will be evaluated while y hasn't been defined yet.  
So former method won't work.

codybartfast

Initially, I too thought both methods would work, I can imagine a 'very lazy' implementation of stream-map the doesn't evaluate its arguments at all until the first element of the stream is requested. But SICP's streams are definately not that lazy as they evaluate the first element of the stream immediately. So y will be evaluated when `(stream-map f y)` is evaluated.

master

But even if that were the case, we would still need to have something to delay. We can't pass an object which doesn't exist to `delay`. That's why it can never work, under any evaluation model.

Note I'm talking here of *evaluation* models, the big problem here is not the lazy evaluation but the scope issue. `let` doesn't allow intercommunication of bindings. One would need to use `let*` or indeed the `define` version.

yd

Previous one will work, the latter don't. Let's trace the latter definition of solve:

First, set both y and dy to \*unassigned\*, that's totally ok. Next, set a by something like

```
(y0 . (delay ..<something involved dy>..))
```

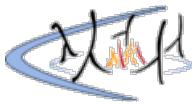
although dy is \*unassigned\*, since it's wrapped by delay, that ok.

Next, we want to set b by the evaluation of `(stream-map f y)`, and here comes the problem.

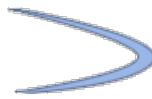
By definition of stream-map, the result will be

```
((f (stream-car y)), (delay (stream-map ....))))
```

but y is \*unassigned\* in current env now, thus evaluation of `(f (stream-car y))` does not run correctly.



# sicp-ex-4.19



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.18) | Index | Next exercise (4.20) >>

anon

These three viewpoints could be laid out on the scale from "imperative" to "declarative". Ben's idea seems to make the most sense, at least for a programmer used to the imperative style. However, it could cause hard to detect bugs, and Scheme is not supposed to be an imperative language anyway. Eva's desired solution seems to be difficult or maybe even impossible to implement, even if it would be kind of nice from a declarative point of view. Alyssa's way of looking at things avoids the problem by simply showing an error and forcing the programmer to write a "better" procedure. This seems to be a good way out.

I don't know how to implement a general system that would make Eva's idea work. For instance, while we could reorder `define's in such a way that `a` comes before `b` (on the grounds that `b` uses `a` in its definition), this would not work if we had to work with a circular dependency (i.e. `b` depends on `a`, `a` depends on `b`).

One way to solve this issue would be to treat every binding as a function, i.e. `b` would be a function of no arguments that returns some value, and the same thing would apply to `a`. Then, evaluation of those values would happen during the call, and the most recent definition of `a` would be used even though no reordering has been done. However, this would fundamentally change how Scheme works.

xdaidiliu

I will describe how to implement Eva's scheme (no pun intended).

One way to do so is to topologically sort the non-function definitions in order of dependency. This can be done by converting the sequence of definitions into a directed graph according to the interdependency of the variables. Meanwhile, we can check the graph for cycles and signal an error if found, and evaluate the definitions in topologically sorted order if no cycles are found.

Needless to say, trying to implement directed acyclic graphs in Scheme, to say nothing of topological sort, is probably non-trivial and may arguably be overkill for implementing a measly little Scheme interpreter.

Hence, here is a conceptually easier way to do it. We first take out all the function definitions and put them at the top, since their bodies are delayed and hence will not pose any issues whatsoever. Then, we take the list of the non-function definitions, generate a list of the matching dependent variables in each body, and repeatedly take out all non-function definitions whose bodies are independent of any remaining non-function variables. This is probably asymptotically slower than topological sort, but it works fine, and has the added bonus of being able to naturally check for cycles.

First, we redefine make-procedure: (note I am assuming all code in chapter 4 up to this exercise has been evaluated, so all the helper functions used by the book to redefine eval is available).

```
(define (function-definition? exp)
  (and (definition? exp)
       (lambda? (definition-value exp)))))

(define (non-function-definition? exp)
  (and (definition? exp)
       (not (function-definition? exp))))

(define (reorder-procedure-body body)
  (let ((func-defs (filter function-definition? body))
        (var-defs (filter non-function-definition? body))
        (non-defs (remove definition? body)))
    (append func-defs
            (reorder-non-function-definitions var-defs)
            non-defs)))

(define (make-procedure parameters body env)
  (list 'procedure
        parameters
        (reorder-procedure-body body)
        env))
```

Then, a few helper functions:

```
;; unrolls nested lists
(define (tree->list tree)
```

```

(if (list? tree)
    (apply-in-underlying-scheme
     append
     (map tree->list tree))
    (list tree))

;; removes duplicates
(define (list->set lst)
  (if (or (null? lst)
          (null? (cdr lst)))
      lst
      (cons (car lst)
            (delete (car lst)
                   (list->set (cdr lst))))))

(define (all-included-symbols symbol-pool seq)
  (intersection-set symbol-pool
    (list->set (tree->list seq))))
;; intersection-set is given in chapter 2 of SICP

;; there are likely faster ways to do this
;; computes set1 - set2 nondestructively
(define (difference-set set1 set2)
  (define (in-set2? obj1)
    (find (lambda (obj2) (eq? obj1 obj2))
          set2))
  (remove in-set2? set1))

```

Finally, the main workhorse function:

```

;; assume no duplicate variables in var-defs, otherwise undefined behavior
(define (reorder-non-function-definitions var-defs)
  (define (no-dependencies? pair)
    (null? (cdr pair)))
  ;; pair here means definition / included symbol pair
  (define (pairs-with-symbols-removed pairs symbols)
    (map (lambda (pair)
           (cons (car pair) (difference-set (cdr pair) symbols)))
         pairs))
  (define (iter pairs-defs-included result)
    (if (null? pairs-defs-included)
        result
        (let ((independent (filter no-dependencies? pairs-defs-included))
              (dependent (remove no-dependencies? pairs-defs-included)))
          (if (null? independent)
              (error "cycle detected in inner non-function defines")
              (let ((symbols-to-remove
                     (map (lambda (pair)
                            (definition-variable (car pair)))
                           independent)))
                  (iter
                   (pairs-with-symbols-removed dependent symbols-to-remove)
                   (append (map car independent) result))))))
  (let* ((symbol-pool (map definition-variable var-defs))
         (pairs-defs-included
          (map (lambda (def)
                 (cons def (all-included-symbols symbol-pool
                                               (definition-value def))))))
         var-defs))
    (reverse (iter pairs-defs-included '())))
  ;; need to reverse because results built using cons, in reverse order

```

Now, assuming that eval is \*not\* the builtin eval but rather the simplified one that has been defined as in the code from the SICP text, here are some examples:

```

(assert (equal? '((define (f x) 7) (define a 3) (define b a))
                (reorder-procedure-body
                 '((define a 3) (define b a) (define (f x) 7)))))

;; example from the exercise
(assert (= 20 (eval '(let ((a 1)
                           (define (f x)
                             (define b (+ a x))
                             (define a 5)
                             (+ a b))
                           (f 10)) the-global-environment)))
       ;; 20, as Eva required.

```

I ran all code above in MIT Scheme.

note we are assuming the bodies of the non-function definitions do not contain redefinitions of variables shared with other non-function definitions. For example, the following should be perfectly legal but may break our program and result in undefined behavior:

```

(define (f)
  (define a 5)
  (define b
    (let ((a 6))
      a))
  b)

```

This is an admitted limitation of our program: to account for this case requires significant further work.

Sphinxsky

A simple method, but does not support recursive definition of variables

```

(define unassigned '*unassigned*)
(define define- 'define-)

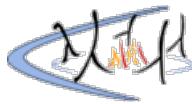
(define (is-define? exp-)
  (tagged-expression? exp- define-))

; fringe is in exercise 2.28
(define (vars-is-contained? vars exp-)
  (define (iter vars exp-list)
    (if (null? vars)
        false
        (if (memq (car vars) exp-list)
            true
            (iter (cdr vars) exp-list))))
  (iter vars (fringe exp-)))

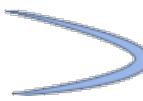
(define (sort-define defines)
  (define (iter self other depend defs vars)
    (if (null? defs)
        (append self other depend)
        (let* ((first (car defs))
               (exp- (definition-value first)))
          (cond ((self-evaluating? exp-)
                  (iter (cons first self) other depend (cdr defs) vars))
                ((vars-is-contained? vars exp-)
                  (iter self other (cons first depend) (cdr defs) vars))
                (else
                  (iter self (cons first other) depend (cdr defs) vars))))))
  (iter '() '() '() defines (map definition-variable defines)))

(define (scan-out-defines proc-body)
  (let ((is-defines (filter is-define? proc-body)))
    (if (null? is-defines)
        proc-body
        (let* ((others (filter
                        (lambda (exp-)
                          (not (is-define? exp-))))
                        proc-body))
          (is-defines (sort-define is-defines))
          (vars (map definition-variable is-defines))
          (vals (map definition-value is-defines))
          (bindings (map
                      (lambda (var)
                        (make-combination var unassigned))
                      vars))
          (sets (map make-set vars vals))
          (new-body (append sets others)))
          (list (make-let bindings new-body)))))))

```



# sicp-ex-4.20



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.19) | Index | Next exercise (4.21) >>

meteorgan

```
; a
;; letrec expression
(define (letrec? expr) (tagged-list? expr 'letrec))
(define (letrec-init expr) (cadr expr))
(define (letrec-body expr) (cddr expr))
(define (declare-variables expr)
  (map (lambda (x) (list (car x) '*unassigned*)) (letrec-init expr)))
(define (set-variables expr)
  (map (lambda (x) (list 'set! (car x) (cadr x))) (letrec-init expr)))
(define (letrec->let expr)
  (list 'let (declare-variables expr)
        (make-begin (append (set-variables expr) (letrec-body expr)))))
```

leafac

```
; The lambda in `let` is evaluated in the context of the enclosing environment,
; in which the bindings of the lambda itself are not in place.

; The trick of encoding `letrec` is that we first establish the bindings, and
; then define the lambdas in an environment where the bindings are there, so
; the recursive call can succeed.

; The following snippets illustrate the difference:

(let ((fact <fact-body>)
      <let-body>)

; is encoded by

((lambda (fact)
  <let-body>)
 <fact-body>

; such that `<fact-body>` can't refer to `fact`. While:

(letrec ((fact <fact-body>)
        <let-body>)

; is encoded by

((lambda (fact)
  (set! fact <fact-body>)
  <fact-body>)
 '*unassigned*)

; note that in the context of `<fact-body>`, the variable `fact` itself is
; bound.
```

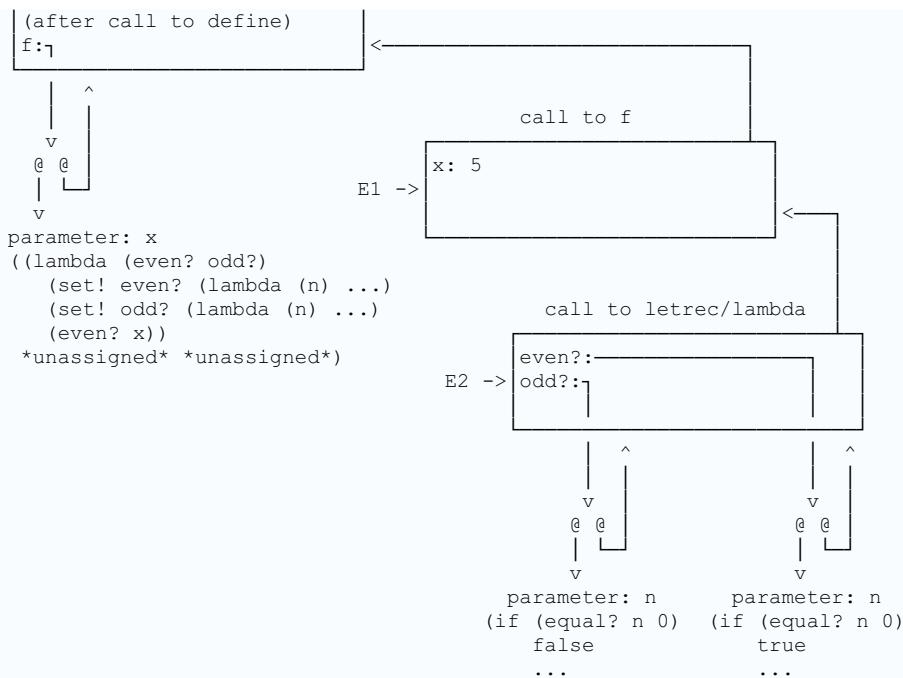
codybartfast

Letrec Environment Diagram

=====

Even? and odd? procs reference E2 because they are created when evaluating set! within the body of the lambda. This means they can lookup the even? and odd? variables defined in this frame.

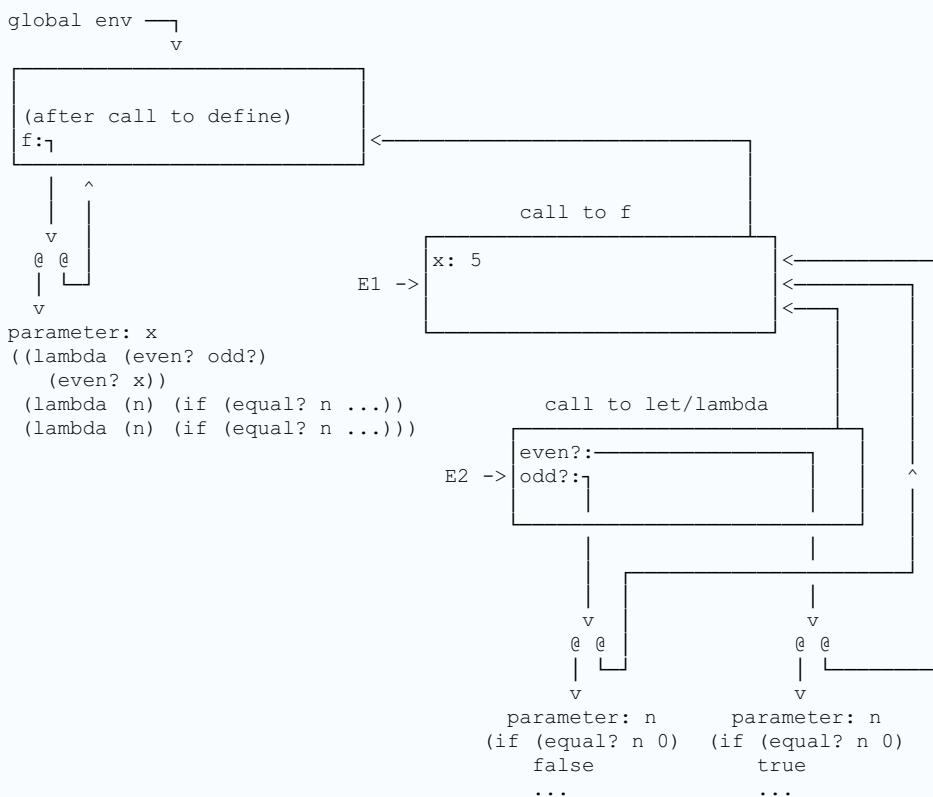
global env └─  
 v



Let Environment Diagram

=====

Even? and odd? procs reference E1 because they are evaluated in the body of f but outside the 'let lambda' because they are passed as arguments to that lambda. This means they can't lookup the even? and odd? variables defined in E2.



squarebat

I implemented the exercise 4.18 form

```

;- we use the form from ex 4.18 because this makes the bindings truly
simultaneous
;- we expect it to throw an error for evaluations like the one in 4.18
;- in essence, cyclic dependencies will only be allowed by letrec if they are wrapped
;by a lambda

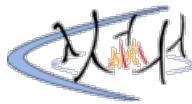
(define (letrec? exp) (tagged-list? exp letrec))
(define (letrec-init? exp) (cadr exp))
(define (letrec-vars exp) (map car (letrec-init? exp)))
(define (letrec-bindings exp) (map cadr (letrec-init? exp)))
(define (letrec-body exp) (caddr exp))

```

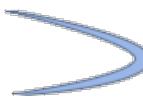
```
(define (get-symbols exp) (map
                           (lambda (x) (next-symbol interpreter-symbol-list))
                           (letrec-initis exp)))
;where interpreter-symbol-list is an infinite stream of randomly generated symbols
(define (letrec->let exp)
  (let ((intermediate-symbols (get-symbols exp)))
    (make-let (map (lambda (var) (list var '*unassigned*)) (letrec-vars exp))
              (list (make-let (map list intermediate-symbols (letrec-bindings exp))
                            (map (lambda (var value) (list 'set! var value)) (letrec-
vars exp) intermediate-symbols))
                    (letrec-body exp))))))
```

---

Last modified : 2021-10-15 05:46:37  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-4.21



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.20) | Index | Next exercise (4.22) >>

meteorgan

```
; a
(lambda (n)
  ((lambda (fib)
    (fib fib n))
   (lambda (f n)
     (cond ((= n 0) 0)
           ((= n 1) 1)
           (else (+ (f f (- n 2)) (f f (- n 1)))))))
  ; b. filling the box as sequence:
(ev? od? (- n 1))
(ev? od? (- n 1))
```

Rptx

```
; a
; this fib procedure uses the iterative version of fib.
((lambda (n)
  ((lambda (fib)
    (fib fib 1 0 n))
   (lambda (fib a b count)
     (if (= count 0)
         b
         (fib fib (+ a b) a (- count 1))))))
  10)
```

SophiaG

As suspected from the recursive process minus recursive function, and confirmed through the footnote, this is a rather roundabout example of Haskell Curry's famous Y Combinator.

I couldn't figure out what they were getting at with the even/odd example, but here's a clearer version of what's going on in the factorial example:

```
;non-recursive factorial function
(define fact-once
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))

;y-combinator
(define Y
  (lambda (f)
    ((lambda (x) (x x))
     (lambda (x) (f (lambda (y) ((x x) y))))))

(define factorial (Y fact-once))
(factorial 20) ;=2432902008176640000
```

xdaividliu

@SophiaG, the odd? even? example is more sophisticated than a simple self-referential recursive function like fibonacci or factorial, since odd? and even? refer \*to each other\*, e.g. are \*mutually\* recursive. Hence, that example is neat because it shows that you can even create \*mutually\* recursive \*sets\* of functions without using define or letrec.

Btw the answer is this

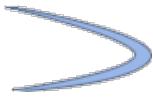
```
(define (f x)
  ((lambda (even? odd?)
    (even? even? odd? x)) ; note above
   (lambda (ev? od? n)
     (if (= n 0) true (od? ev? od? (- n 1))))
   (lambda (ev? od? n)
     (if (= n 0) false (ev? ev? od? (- n 1)))))))
```

---

Last modified : 2019-05-08 06:10:05  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-4.22



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.21) | Index | Next exercise (4.23) >>

meteorgan

```
; ; add this in function analyze  
((let? expr) (analyze (let->combination expr)))
```

shifuda

For the sake of completion, adding the let->combination functions (which is already there in exercise 4.6)

```
(define (let? exp) (tagged-list? exp 'let))  
  
(define (let-variables exp)  
  (define (internal lis)  
    (if (null? lis) '()  
        (cons (caar lis) (internal (cdr lis))))  
    ))  
  
  (internal (cadr exp))  
 )  
  
(define (let-body exp)  
  (cddr exp))  
  
(define (let-expressions exp)  
  (define (internal lis)  
    (if (null? lis) '()  
        (cons (cadar lis) (internal (cdr lis))))  
    ))  
  
  (internal (cadar exp))  
 )  
  
(define (make-lambda-combination lambda-dec expressions)  
  (cons lambda-dec expressions)  
 )  
  
(define (let->combination exp)  
  (make-lambda-combination  
    (make-lambda (let-variables exp) (let-body exp))  
    (let-expressions exp)))  
 )  
  
;; (let->combination '(let ((var1 exp1) (var2 exp2)) (body)))  
;; ;Value: ((lambda (var1 var2) (body)) exp1 exp2)
```

Sphinxsky

```
(define (let? exp-)  
  (tagged-list? exp- 'let-))  
  
(define (let-body exp-)  
  (cddr exp-))  
  
(define (let-variables exp-)  
  (cadar exp-))
```

```

(define (consortium-variable consortium)
  (car consortium))
(define (consortium-value consortium)
  (cadr consortium))
(define (separate variables)
  (define (iter variables variable value)
    (if (null? variables)
        (cons (reverse variable) (reverse value)))
        (let ((first (car variables)))
          (iter
            (cdr variables)
            (cons (consortium-variable first) variable)
            (cons (consortium-value first) value))))))
  (iter variables '() '()))
(define (analyze-let exp-)
  (let* ((vars-vals (separate (let-variables exp-)))
         (lambda-exp (make-lambda (car vars-vals) (let-body exp-)))
         (ana-lambda (analyze-lambda lambda-exp))
         (args (map analyze (cdr vars-vals))))
    (lambda (env)
      (execute-application
        (ana-lambda env)
        (map (lambda (arg) (arg env)) args)))))

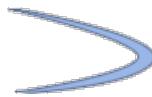
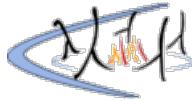
; add in analyze
((let? exp-) (analyze-let exp-))

```

---

Last modified : 2020-09-08 15:26:21  
 WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-4.23



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.22) | Index | Next exercise (4.24) >>

meteorgan

In Alyssa's analyze-sequence, execute-sequence is running in runtime. But the solution in the text unrolls this procedure. for example: for `((lambda (x) (+ x 1)) 1)`, Alyssa's analyze-sequence is `(lambda (env) (execute-sequence (lambda ... env)))`, analyze-sequence in text is `(lambda (env) ((lambda ...) env))`.

SophiaG

This finally seems to be at least a vague reference to interpreters as iterating down a tree structure. It seems pretty clear from comparison that the version of analyze-sequence presented in the text cdrs down \*two\* branches of a procedure's environment for every run through its "loop" procedure, whereas Alyssa's only cdrs to the end of one possible sequence and executes. Thus its efficiency gains over the original eval grow quadratically with the complexity of the procedure it analyzes.

poly

I think the Alyssa's version is less efficient than the text's when there are function calls. In the text's version, the analyze-sequence will return a lambda procedure like

```
(lambda (env)
  ((lambda (env) (proc1 env) (proc2 env))
   env)
  (proc3 env))
```

It's already sequentialized. When it is applied to a env argument, it will apply all the procs to the env.

In the Alyssa's version, the result would be like

```
(lambda (env (executive-sequence procs env)))
```

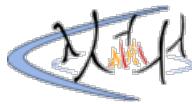
The sequential procedure is inside of the executive-sequence, which means there will be extra work whenever there is a function call.

Owen

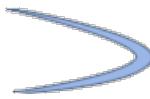
I agree with poly. I think the overhead of Alyssa's version are

1. Cost of traversing the list of procedures.

2. Cost of condition checking when executing each procedure



# sicp-ex-4.24



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.23) | Index | Next exercise (4.25) >>

felix021

Below are 2 simple tests, loops and calculating fibonacci numbers.

```
; 4-24.test1.scm

(define (loop n)
  (if (> n 0)
      (loop (- n 1)))

(loop 1000000)
(exit)
```

```
; 4-24.test2.scm

(define (fib n)
  (if (<= n 2)
      1
      (+ (fib (- n 1)) (fib (- n 2)))))

(fib 30)
(exit)
```

My results are:

only eval: loop 9.689s, fib 18.880s

analyze + eval: loop 5.528s, fib 10.682s

master

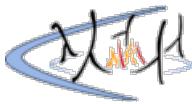
I wanted to *really* put our interpreter to the test and see how long it would take it to compute A(4,1), i.e. the Ackermann function and I believe the highest value which is practical to compute. Well, after 40 minutes of waiting for the slow interpreter to finish I decided that it was not going to happen anytime soon. Therefore, I made it compute A(3,8) instead, and after that A(3, 10), which was as high as I was willing to go. Here's some data, including benchmarks from Chez Scheme for reference (which I've heard is the fastest Scheme implementation, it's certainly up there):

```
(A 3 8)
Slow: ~26.47s
Fast: ~15.06s
Chez Scheme: basically instantaneous

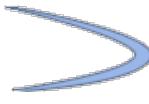
(A 3 10)
Slow: ~6m32.40s
Fast: ~3m33.66s
Chez Scheme: basically instantaneous

(A 4 1)
Slow: at least 40m
Fast: at least 15m (probably much more, wasn't willing to wait longer)
Chez Scheme: ~5.25s
```

Our interpreter is obviously **extremely** inefficient. But the extra work at "compile time" really pays off, the analyzing interpreter is roughly twice as fast.



# sicp-ex-4.25



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.24) | Index | Next exercise (4.26) >>

meteorgan

In applicative-order Scheme, when call (factorial 5), the call will **not** end. because, when call unless, even **if** (= n 1) is true, (factorial (- n 1)) will be called. so n will be 5, 4, 3, 2, 1, 0, -1 .... . In normal-order Scheme, this will work, Because normal-order Scheme uses lazy evaluation, when (= n 1) is true, (factorial n) will **not** be called.

Anonymous

Here is an implementation in the underlying scheme to see that it indeed works this way.

```
(define-macro (unless-lazy predicate action alternative)
  `(if (not ,predicate) ,action ,alternative))

(define (unless-applicative predicate action alternative)
  (if (not predicate) action alternative))

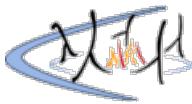
(define (factorial n)
  (unless-applicative (= n 1)
    (* n (factorial (- n 1)))
    1))

(factorial 5)
```

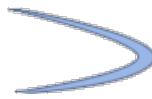
krubar

In "normal order" (as described in first chapter) Scheme this procedure should also loop forever. Since normal order does not necessarily means lazy.

So normal order evaluation expands procedures and arguments to most primitive form and only then evaluate it. In this case evaluator would keep expanding (factorial (- n 1)) forever.



# sicp-ex-4.26



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.25) | Index | Next exercise (4.27) >>

meteorgan

```
; ; add this code in eval
((unless? expr) (eval (unless->if expr) env))

;; unless expression is very similar to if expression.

(define (unless? expr) (tagged-list? expr 'unless))
(define (unless-predicate expr) (cadr expr))
(define (unless-consequence expr)
  (if (not (null? (cdddr expr)))
      (cadddr expr)
      'false))
(define (unless-alternative expr) (caddr expr))

(define (unless->if expr)
  (make-if (unless-predicate expr) (unless-consequence expr) (unless-alternative expr)))
```

SophiaG

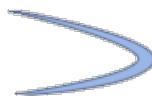
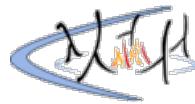
First I should note that my implementation (same as meteorgan's above) does not even work for the lazy factorial procedure...it aborts due to infinite recursion. As for actual higher order procedures here's a very clever example borrowed from **Xueqiao Xu**, binary pattern matching:

```
(define select-y '#t #f #t #t)
(define xs '(1 3 5 7))
(define ys '(2 4 6 8))
(define selected (map unless select-y xs ys))
```

Richard

It actually works well with the factorial procedure in my implementation.

# sicp-ex-4.27



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.26) | Index | Next exercise (4.28) >>

x davidiu

The strange behavior of the variable count happens not because of the first expression in the body of id (e.g. "identity"), but rather the \*second\*. To see what I mean, let's first examine the example from the exercise. In order to provide more clarity, I will call eval and actual-value directly as opposed to using the driver-loop REPL provided in the book.

```
(let ((env the-global-environment))
  (eval '(define count 0) env)
  (eval '(define w (id (id 10))) env)
  (assert (= 1 (eval 'count env)))
  (assert (= 10 (actual-value 'w env)))
  (assert (= 2 (eval 'count env))))
```

Here, the 10 and 2 are trivial and require no explanation. The 1 is most interesting: it is because when we eval the expression (id (id 10)) in env when we are first defining w, the procedure list-of-delayed-args has x in the body of id bound to the result of (delay-it '(id 10) env), a thunk. Since (eval-definition exp env) calls eval, not actual-value, on (definition-value exp), it is the thunk that is bound to w, not the number 10. That thunk is not forced until we call actual-value on w, and hence before doing that, count has been incremented only once.

Note we have used eval for the assert lines involving count, since count is always bound to a number, but I have used actual-value for the assert involving w, since at that moment w is bound to a thunk and must be forced. If we had tried to call eval on w immediately after defining it, eval would return the thunk object, and attempting to print it may result in an infinite recursion loop since the thunk object contains env, which is a list that may contain cycles.

Interestingly, if we slightly modify the definition of id, we actually get different behavior:

```
(eval '(define (id2 x)
           (set! count (plus count 1))
           (+ 0 x)) ; note (+ 0 x) as opposed to x
           the-global-environment)

(let ()
  (eval '(define count 0) the-global-environment)
  (eval '(define w (id2 (id2 10))) the-global-environment)
  (assert (= 2 (eval 'count the-global-environment))) ; note 2 not 1!
  (assert (= 10 (actual-value 'w the-global-environment)))
  (assert (= 2 (eval 'count the-global-environment))))
```

Here, the expression (+ 0 x), unlike the expression x, immediately gets forced, even though x is bound to a thunk, since + is a primitive procedure. Hence, the strange behavior of count is \*not\* due to the set! part of id, but rather the x part!

canardivore

```
(define w (id (id 10)))

;; L-Eval input:
count
;; L-Eval value:
0
Defining w doesn't evaluate it.
;; L-Eval input:
w
;; L-Eval value:
10
now that w is in the prompt, w is forced to evaluate, evaluating both ids. so now w = 10,
count = 2.
;; L-Eval input:
count
;; L-Eval value:
2
```

uuu

I got different result, which is very interesting

```
;;; L-Eval input:
(define count 0)
;;; L-Eval value:
ok
;;; L-Eval input:
(define (id x) (set! count (+ count 1)) x)
;;; L-Eval value:
ok
;;; L-Eval input:
(define w (id (id 10)))
;;; L-Eval value:
ok
;;; L-Eval input:
count
;;; L-Eval value:
1
**It's not zero here**
;;; L-Eval input:
w
;;; L-Eval value:
10
;;; L-Eval input:
count
;;; L-Eval value:
2
**changed after "w"**
;;; L-Eval input:
w
;;; L-Eval value:
10
;;; L-Eval input:
count
;;; L-Eval value:
3
**changed after "w", again!**
;;; L-Eval input:
w
;;; L-Eval value:
10
;;; L-Eval input:
count
;;; L-Eval value:
4
**changed after "w", one more time!**
;;; L-Eval input:
```

poly

Actually, the value of count will never change no matter how many times you evaluate 'w' (at least once), if you have a memoized thunk.

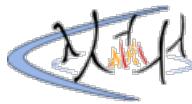
revc

Three straightforward diagrams:

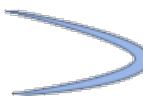


```
+-----+
| count: 2
| id: (lambda (x) (set! ...)) <--> #global environment#
global---+ w: (evaluated-thunk 10 #global environment#) | w
| |
| +-----+
```

newone	( <b>define</b> env the-global-environment) (actual-value '( <b>define</b> count 0) env) (actual-value '( <b>define</b> (id x) ( <b>set!</b> count (+ count 1)) x) env) (actual-value '( <b>define</b> w (id (id 10))) env)  (actual-value 'count env) ; 1 (actual-value 'w env) ;10 (actual-value 'count env) ;2
--------	--



# sicp-ex-4.28



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.27) | Index | Next exercise (4.29) >>

meteorgan

for example:  
`(define (g x) (+ x 1))  
(define (f g x) (g x))`

when call (f g 10), `if` don't use actual-value which will call force-it, g will be passed as parameter which will be delayed, then g is a thunk, can't be used as function to call 10.

poly

Let me make meteorgan's answer more specific. 'g' will be passed as a parameter which will be delayed literally. It means 'g' is a thunk and (g 10) will be considered as application in the 'eval'. In the procedure apply, the 'g' will be seen as a procedure, with a tag 'thunk', which is not a primitive procedure and compound procedure. So the procedure apply will report an error.

Sphinxsky

a simpler example

```
(define (proc operate) operate)
; error: Unknown procedure type -- APPLY (thunk + (...))
((proc +) 1 2)
```

ryx

Could anyone provide more detailed explanation for this?

I understand neither the context nor the intention of this code, but except for the comment it seems perfectly fine:

```
$ racket
Welcome to Racket v7.9 [bc].
> (define (proc x) x)
> ((proc +) 1 2)
3
>
```

The comment is more sloppy than wrong; apply works as expected

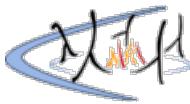
```
> (apply (proc +) (list 1 2))
3
>
```

although why apply is of interest is unclear. The unknown procedure type error is mysterious both in its meaning and to what it is referring. Thunk is being misused; generally construed, a thunk is a parameterless procedure, so it is unnecessary, although possible, to call one via apply

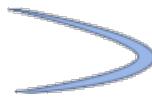
```
> (define (one) 1)
> (apply one '())
1
> (one)
1
>
```

What in particular do you feel requires a more detailed explanation?





# sicp-ex-4.29



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.28) | Index | Next exercise (4.30) >>

x davidiu

Here's a very simple, trivial example that illustrates the difference between memoizing and not memoizing thunks. First, assume that the force-it procedure has been defined with memoization, as in the longer of the two versions of force-it given in the book.

We also define a non-memoized version, which is simply the shorter of the two versions of force-it given in the book:

```
(define (unmemoized-force-it obj)
  (if (thunk? obj)
      (actual-value (thunk-exp obj) (thunk-env obj))
      obj))
```

Here is our simple example, which can be run in MIT Scheme:

```
(eval '(define (identity x) x) the-global-environment)

(eval '(define (identity-with-computation x)
         (display "1 hour elapsed ")
         x)
      the-global-environment)

(begin ; with memoization
  (eval '(define z (identity (identity-with-computation 0)))
        the-global-environment)
  (actual-value 'z the-global-environment)
  (actual-value 'z the-global-environment))
  ; displays "1 hour elapsed" *once*

  ; fluid-let is an MIT Scheme special form for dynamic (as opposed to lexical for ordinary
  let) binds

  (fluid-let ((force-it unmemoized-force-it)) ; without memoization
    (eval '(define z (identity (identity-with-computation 0)))
          the-global-environment)
    (actual-value 'z the-global-environment)
    (actual-value 'z the-global-environment))
    ; displays "1 hour elapsed" *twice*
```

What's happening here is that the argument given to identity, the expression '(identity-with-computation 0), is stored in a thunk via delay-it. With memoization, that thunk is effectively forced only once, but without it, that thunk is forced every time we call actual-value on z.

Note that memoization of thunks is similar, but \*not exactly\* the same thing as the memoization you may be used to in dynamic programming (the canonical example being the naive recursive algorithm for fibonacci, which is  $O(\exp N)$  without dynamic programming (DP) memoization but  $O(N)$  \*with\* DP memoization).

In DP memoization, a function  $f(x)$  is computed \*once\* for every  $x$ ; further calls to  $f(x)$  for the same  $x$  will simply return the stored value. For the thunk memoization discussed in this section of SICP, memoization is done for each \*thunk object\*, regardless of whether the arguments are the same.

For example:

```
(begin ; with thunk memoization
  (actual-value '(identity-with-computation 0) the-global-environment)
  (actual-value '(identity-with-computation 0) the-global-environment))
  ; displays "1 hour elapsed" *twice*, not once, even though both calls had the same
  argument 0. This is because both calls generated *separate* thunk objects. Thunk
  memoization does *not* help in this case!
```

The takeaway here is that thunk memoization does \*not\* tabulate previously computed results by \*argument\*, the way dynamic programming memoization does. Note that DP memoization was actually implemented earlier in the book (chapter 3 if I remember correctly).

Hence, I would personally caution against using the  $O(\exp N)$  naive recursive fibonacci algorithm for this exercise, since in that case, the multiple redundant calls of (fib n) to the same n are \*not\* sharing work, but are \*still\* redundantly doing work since they result in separate thunk objects. Even if thunk

memoization \*does\* make a difference here, it is certainly not nearly as much as the jump from  $O(\exp N)$  to  $O(N)$  that true dynamic programming memoization, which involves actually \*tabulating computed results indexed by argument\*.

```
felix021 ;  
;  
; a) with memoization: 0.43s, without memoization: 9.3s  
(define (fib i)  
  (if (<= i 2)  
      1  
      (+ (fib (- i 1)) (fib (- i 2)))))  
  
(define (test x)  
  (define (iter t)  
    (if (= t 0)  
        0  
        (+ x (iter (- t 1)))))  
  (iter 10))  
  
(test (fib 20))  
  
(exit)
```

meteorgan

```
with memoization:  
(square (id 10))  
=> 100  
count  
=>1  
  
without memoization:  
(square (id 10))  
=>100  
count  
=>2
```

revc

a simple explanation by diagrams:

```
+-----+  
| global environment |  
+-----+  
  ^  
  |  
+-----+  
| x: #Thunk:(id 10) | (square (id 10))  
+-----+  
  
(* x x) --> (* (actual-value x) (actual-value x))  
          (* (force-it #Thunk) (force-it #Thunk))  
          (* (id 10) (id 10)) ;;set count twice
```

pvk

With memoization, the first call to force-it replaces the binding of `x`, ('`thunk (id 10) <env>`), with an evaluated thunk, ('`evaluated-thunk 10`), incrementing count once.  
The second call to force-it just gets the value of the evaluated thunk. This is why the memoized version only increments count once.

shifuda

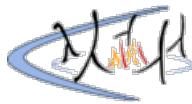
```
(define (f1 x) x)  
(define f2 (f1 f1))  
  
(define t1 (real-time-clock))  
(define (loop i n)  
  (if (= i n) '()
```

```
(begin
  (f1 2)
  (f2 2)
  (loop (+ 1 i) n))))
(loop 1 100)
(p (- (real-time-clock) t1))

;; with memoization the above runs in 55ms
;; without memoization, it runs in 300ms
```

---

Last modified : 2023-07-27 15:07:52  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-4.30

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.29) | Index | Next exercise (4.31) >>

meteorgan

```
;; a
In begin expression, every expression will be evaluated using eval, and display is primitive function, it will call force-it to get x.

;; b
original eval-sequence:
(p1 1) => (1 . 2)
(p2 1) => 1 . because (set! x (cons x '(2))) will be delayed, in function p, when evaluating it, it's a thunk.

Cy's eval-sequence:
(p1 1) => (1 . 2)
(p2 1) => (1 . 2). thunk (set! x (cons x '(2))) will be forced to evaluate.

;; c
when using actual-value, it will call (force-it p), if p is a normal value, force-it will return p, just as never call actual-value

;; d
I like Cy's method.
```

verdammelt

;; d

I prefer the original style. In my opinion (**and** this is an opinion question), a normal order interpreter should ONLY **force** a thunk that is needed. Since only the final value of the sequence is used (returned) the others are **not** needed **and** should **not** be forced. This is for consistency.

Igessler

For d, I feel like the answer to this question hinges on whether this claim is true:

All items in a sequence except for the last one are only there for the side effects they produce.

This seems like it should be true, because it's easy to see that the value of an expression in non-final position in a sequence is discarded. If this is true, then I think we must prefer Cy's method, as the only sane reason for inclusion of a non-final sequence item would be for its side-effects.

newone

b.

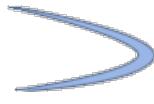
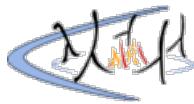
`(p2 1)` returns a thunk, which is a loop containing the current environment.

d.

I prefer evaluating a variable when it is referenced. I think everytime a variable is referenced, it's value is needed.

```
((variable? exp)
 (force-it (lookup-variable-value exp env)))
```

# sicp-ex-4.31



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.30) | Index | Next exercise (4.32) >>

woofy

```
(define (apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure
          procedure
          (list-of-arg-values arguments env)))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameter-names procedure)
           (compound-procedure-args procedure arguments env)
           (procedure-environment procedure))))
        (else
         (error "Unknown procedure type -- APPLY" procedure)))))

(define (lazy-param? param) (eq? 'lazy (cadr param)))
(define (lazy-memo-param? param) (eq? 'lazy-memo (cadr param)))
(define (eager-param? param) (symbol? param))

(define (compound-procedure-args procedure arguments caller-env)
  (define (build-list params arg-expss)
    (define (build param exp)
      (cond ((eager-param? param) (actual-value exp caller-env))
            ((lazy-param? param) (delay-it exp caller-env))
            ((lazy-memo-param? param) (delay-it-memo exp caller-env))
            (else (error "Invalid parameter specification -- COMPOUND-PROCEDURE-ARGS" param)))
      (map build params arg-expss)))
  (build-list (procedure-parameters procedure) arguments))

(define (actual-value exp env)
  (force-it (eval exp env)))

(define (delay-it exp env) (list 'thunk exp env))
(define (delay-it-memo exp env) (list 'thunk-memo exp env))
(define (thunk? obj) (tagged-list? obj 'thunk))
(define (thunk-memo? obj) (tagged-list? obj 'thunk-memo))
(define (thunk-exp thunk) (cadr thunk))
(define (thunk-env thunk) (caddr thunk))
(define (evaluated-thunk? obj) (tagged-list? obj 'evaluated-thunk))
(define (thunk-value evaluated-thunk) (cadr evaluated-thunk))

(define (force-it obj)
  (cond ((thunk? obj) (actual-value (thunk-exp obj) (thunk-env obj)))
        ((thunk-memo? obj)
         (let ((result (actual-value (thunk-exp obj) (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)
           (set-cdr! (cdr obj) '())
           result))
        ((evaluated-thunk? obj)
         (thunk-value obj))
        (else obj)))

(define (procedure-parameter-names p)
  (map (lambda (x) (if (pair? x) (car x) x)) (procedure-parameters p)))
```

Felix021

```
(include "4.2.2.scm") ;;comment (driver-loop) in 4.2.2.scm
;; omit the "lazy-memo" requirement for simplicity...
```

```

(define (apply procedure arguments env)
  (cond
    ((primitive-procedure? procedure)
     (apply-primitive-procedure
      procedure
      (list-of-arg-values arguments env)))
    ((compound-procedure? procedure)
     (eval-compound-procedure procedure arguments env))
    (else
     (error "Unknown procedure type -- APPLY" procedure)))))

(define (eval-compound-procedure procedure arguments env)
  (define (iter-args formal-args actual-args)
    (if (null? formal-args)
        '()
        (cons
         (let ((this-arg (car formal-args)))
           (if (and (pair? this-arg)
                     (pair? (cdr this-arg))) ; avoid error if arg is
               ; 1 element list.
               (eq? (cadr this-arg) 'lazy))
               (delay-it (car actual-args) env)
               ;force the argument if it is not lazy.
               (actual-value (car actual-args) env)))
         (iter-args (cdr formal-args) (cdr actual-args)))))

  (define (procedure-arg-names parameters)
    (map (lambda (x) (if (pair? x) (car x) x)) parameters))

  (eval-sequence
   (procedure-body procedure)
   (extend-environment
    (procedure-arg-names (procedure-parameters procedure))
    (iter-args
     (procedure-parameters procedure)
     arguments)
    (procedure-environment procedure)))))

(driver-loop)

;; test ;;

;
; M-Eval input:
;(define x 1)
;
; M-Eval value:
;ok
;
; M-Eval input:
;(define (p (e lazy)) e x)
;
; M-Eval value:
;ok
;
; M-Eval input:
;(p (set! x (cons x '(2))))
;
; M-Eval value:
;1
;
; M-Eval input:
;(exit)
;

```

atupal	<pre> ; start Exercise 4.31 (define (procedure-parameters-ex4.31 p)   (define (name parameter)     (if (pair? parameter)         (car parameter)         parameter))   (define (parameter-names parameters)     (if (null? parameters)         '()         (cons (name (car parameters))               (parameter-names (cdr parameters))))))   (parameter-names (cadr p))) (define (procedure-raw-parameters p) (cadr p))  (define (apply-ex4.31 procedure arguments env)   (cond [(primitive-procedure? procedure)          (apply-primitive-procedure </pre>
--------	---

```

procedure
  (list-of-arg-values arguments env) ) ] ; changed
((compound-procedure? procedure)
  (eval-sequence
    (procedure-body procedure)
    (extend-environment
      (procedure-parameters procedure)
      (list-of-delayed-args (procedure-raw-parameters procedure) arguments env)
      (procedure-environment procedure))) ; changed
  (else (error "Unknow procedure type: APPLY"
    procedure)))))

(define (list-of-delayed-args-ex4.31 raw_parameters exps env)
  (define (arg-value raw_parameter exp)
    (if (pair? raw_parameter)
        (cond ((eq? (cadr raw_parameter) 'lazy)
               (delay-it-no-memo exp env))
              ((eq? (cadr raw_parameter) 'lazy-memo)
               (delay-it exp env))
              (else (error "Unknow parameter type LIST-OF-DELAYED-ARGS:" (cadr
raw_parameter))))
            (actual-value exp env)))
        (if (no-operands? exps)
            '()
            (cons (arg-value (car raw_parameters)
                  (first-operand exps))
              (list-of-delayed-args-ex4.31 (cdr raw_parameters)
                  (rest-operands exps)
                  env)))))

(define (delay-it-no-memo exp env)
  (list 'thunk-no-memo exp env))
(define (thunk-no-memo? obj)
  (tagged-list? obj 'thunk-no-memo))

(define (force-it-ex4.31 obj)
  (cond ((thunk? obj)
         (let ((result (actual-value (thunk-exp obj)
                         (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj)
             result) ; replace exp with its value
           (set-cdr! (cdr obj)
             '())
           result)
         ((evaluated-thunk? obj) (thunk-value obj))
         ((thunk-no-memo? obj) (actual-value (thunk-exp obj)
                         (thunk-env obj)))
         (else obj))) ; forget unneeded env

(define apply apply-ex4.31)
(define force-it force-it-ex4.31)
(define procedure-parameters procedure-parameters-ex4.31)
(define list-of-delayed-args list-of-delayed-args-ex4.31)

;(define (id x)
;  (set! count (+ count 1)) x)
;
; (define count 0)
;
; (define (square x) (* x x))
;
; (square (id 10))
;
; count ; 1
;
; (define count 0)
;
; (define (square (x lazy)) (* x x))
;
; (square (id 10))
;
; count ; 2
;
; (define count 0)
;
; (define (square (x lazy-memo)) (* x x))
;
; (square (id 10))
;
; count ; 1
; end exercise 4.31

```

poly

I prefer to implement the lazy and lazy-memo as expressions, and the thunk selection funtions still work because the lazy and lazy-memo are still thunk, which I just change its tag.

```

;; the expression of lazy and lazy-memo
(define (lazy-parameter? p)
  (and (pair? p) (eq? (cadr p) 'lazy) (null? (cddr p))))

(define (lazy-memo-parameter? p)
  (and (pair? p) (eq? (cadr p) 'lazy-memo) (null? (cddr p))))

(define (lazy? obj)
  (tagged-list? obj 'lazy))

(define (lazy-memo? obj)
  (tagged-list? obj 'lazy-memo))

(define (eval-lazy-memo? obj)
  (tagged-list? obj 'eval-lazy-memo))

(define (delay-lazy exp env)
  (list 'lazy exp env))

(define (delay-lazy-memo exp env)
  (list 'lazy-memo exp env))

(define (force-it obj)
  (cond ((lazy? obj)
         (actual-value (thunk-exp obj) (thunk-env obj)))
        ((lazy-memo? obj)
         (let ((result (actual-value (thunk-exp obj)
                                      (thunk-env obj))))
           (set-car! obj 'eval-lazy-memo)
           (set-car! (cdr obj) result)
           (set-cdr! (cdr obj) '())
           result))
        ((eval-lazy-memo? obj)
         (thunk-value obj))
        (else obj)))

(define (actual-value exp env)
  (force-it (eval exp env)))

;; change some details
(define (apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure
          procedure
          (list-of-arg-values arguments env)))
        ((compound-procedure? procedure)
         (let ((parameters (procedure-parameters procedure)))
           (eval-sequence
            (procedure-body procedure)
            (extend-environment
             (rib-statements parameters) ; changed
             (list-of-delayed-args parameters arguments env) ; 
             (procedure-environment procedure))))))
        (else
         (error "Unknown procedure type -- APPLY" procedure)))))

(define (rib-statements parameters)
  (if (null? parameters)
      '()
      (let ((first (car parameters)) (rest (cdr parameters)))
        (cond ((or (lazy-parameter? first)
                   (lazy-memo-parameter? first))
               (cons (car first) (rib-statements rest)))
              ((variable? first)
               (cons first (rib-statements rest)))
              (else
               (error "Bad Syntax" first))))))

(define (list-of-delayed-args paras exps env) ; changed
  (if (no-operands? exps)
      '()
      (cons (cond ((lazy-parameter? (car paras))
                   (delay-lazy (first-operand exps) env))
                  ((lazy-memo-parameter? (car paras))
                   (delay-lazy-memo (first-operand exps) env))
                  (else
                   (eval (first-operand exps) env)))
             (list-of-delayed-args (cdr paras) (rest-operands exps) env)))))


```

the other things remain the same.

The following procedures were written by modifying existing procedures or adding new procedures. I will mark it (modified? or added?) with comments.

```

;::::::::::::::::::;
;::::: for mceval.scm::::;;
;::::::::::::::::::;

(define (procedure-parameters p) (map (lambda (x) (if (pair? x) (car x) x)) (cdr p))) ;;
modified

;; return a list of keywords consisting of three elements(active, lazy or lazy-memo)
(define (procedure-keywords p) (map (lambda (x) (if (pair? x) (cdr x) 'active)) (cdr p))) ;;; added

;::::::::::::::::::;
;::::: for leval.scm::::;;
;::::::::::::::::::;

(define (apply procedure operands env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure
          procedure
          (list-of-arg-values operands env)))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-keyworded-args procedure operands env) ; changed
           (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure)))) ;; modified

;; return a list of arguments in which the operation to an operand depends on its
corresponding keyword, which is either transforming it into a thunk or evaluating its value
directly.

(define (list-of-keyworded-args procedure exps env)
  (let loop ([keywords (procedure-keywords procedure)]
            [operands exps]
            [reversed-keyworded-args '()])
    (cond [(null? keywords) (reverse reversed-keyworded-args)]
          [(eq? (car keywords) 'lazy) (loop (cdr keywords)
                                             (cdr operands)
                                             (cons (delay-it-nonmemo (first-operand
operands) env)
                                                   reversed-keyworded-args))]
          [(eq? (car keywords) 'lazy-memo) (loop (cdr keywords)
                                              (cdr operands)
                                              (cons (delay-it-memo (first-operand
operands) env)
                                                    reversed-keyworded-args))]
          [(eq? (car keywords) 'active) (loop (cdr keywords)
                                              (cdr operands)
                                              (cons (actual-value (first-operand
operands) env)
                                                    reversed-keyworded-args))]
          [else (error "Unknown keyword" (car keywords))]))]) ;; added

;; Representing thunks

;; non-memoizing version of force-it

(define (force-it-nonmemo obj)
  (actual-value (thunk-exp obj) (thunk-env obj))) ;; added

;; memoizing version of force-it
(define (force-it-memo obj)
  (let ((result (actual-value
                 (thunk-exp obj)
                 (thunk-env obj))))
    (set-car! obj 'evaluated-thunk)
    (set-car! (cdr obj) result) ; replace exp with its value
    (set-cdr! (cdr obj) '()) ; forget unneeded env
    result)) ;; added

;; thunks

(define (delay-it-memo exp env)
  (list 'thunk-memo exp env)) ;; added

(define (delay-it-nonmemo exp env)
  (list 'thunk-nonmemo exp env)) ;; added

```

```

(define (thunk-memo? obj)
  (tagged-list? obj 'thunk-memo)) ;; added

(define (thunk-nonmemo? obj)
  (tagged-list? obj 'thunk-nonmemo)) ;; added

;; generalized version of force-it

(define (force-it obj)
  (cond ((thunk-memo? obj) (force-it-memo obj))
        ((thunk-nonmemo? obj) (force-it-nonmemo obj))
        ((evaluated-thunk? obj) (thunk-value obj))
        (else obj)))
;;;;;;;;
;;;;;; test;;;;;;
;;;;;;;

;;;;;;
;;;;;lazy;;;
;;;;;;

;; L-Eval input:
(define (try a (b lazy)) (if (= a 0) 1 b))

;; L-Eval value:
ok

;; L-Eval input:
(try 0 1)

;; L-Eval value:
1

;; L-Eval input:
(try 0 (/ 1 0))

;; L-Eval value:
1

;;;;;;
;;;;; lazy-memo;;;
;;;;;;;

;; L-Eval input:
(define (try a (b lazy-memo)) (if (= a 0) 1 b))

;; L-Eval value:
ok

;; L-Eval input:
(try 0 (/ 1 0))

;; L-Eval value:
1

;;;;;;
;;;;; non-lazy;;;
;;;;;;;

;; L-Eval input:
(define (try a b) (if (= a 0) 1 b))

;; L-Eval value:
ok

;; L-Eval input:
(try 0 (/ 1 0))
Exception in /: undefined for 0

```

Sphinxsky

Modification based on 4.2.2 in the book

```

(define normal 'normal)
(define lazy 'lazy)
(define lazy-memo 'lazy-memo)

```

```

(define (is-lazy-memo? thunk)
  (tagged-list? thunk lazy-memo))

(define (is-lazy? thunk)
  (tagged-list? thunk lazy))

(define (type-arg arg)
  (if (pair? arg)
      (let ((type (cadr arg)))
        (cond ((eq? type lazy) lazy)
              ((eq? type lazy-memo) lazy-memo)
              (else (error "Unknown parameter type -- TYPE-ARG" type))))
      normal))

(define (get-arg arg)
  (if (pair? arg)
      (car arg)
      arg))

(define (list-of-delayed-args types exps env)
  (if (no-operands? exps)
      '()
      (let ((first (first-operand exps))
            (type (car types)))
        (cons
          (if (eq? normal type)
              (eval- first env)
              (cons type (delay-it first env)))
          (list-of-delayed-args (cdr types) (rest-operands exps) env)))))

(define (apply- procedure arguments env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure
          procedure
          (list-of-arg-values arguments env)))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (map get-arg (procedure-parameters procedure))
           (list-of-delayed-args
            (map type-arg (procedure-parameters procedure))
            arguments
            env)
           (procedure-environment procedure))))
         (else (error "Unknown procedure type -- APPLY" procedure)))))

(define (force-it-memo obj)
  (cond ((thunk? obj)
         (let ((result (actual-value
                       (thunk-exp obj)
                       (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)
           (set-cdr! (cdr obj) '())
           result)
         ((evaluated-thunk? obj)
          (thunk-value obj))
         (else obj)))

(define (actual-value exp- env)
  (let ((result (eval- exp- env)))
    (cond ((is-lazy-memo? result)
           (force-it-memo (cdr result)))
          ((is-lazy? result)
           (force-it (cdr result)))
          (else result)))

```

pvk

Ordinary procedure definitions bind names to lambda expressions, which are then evaluated to procedure objects. I make a definition of the form:

```
(define (foo x (y lazy) (z lazy-memo)) ...)
```

bind `foo` to a "lazy-lambda" expression of the form

```
(lazy-lambda (x y z) (strict lazy lazy-memo) ...)
```

which evaluates to a "lazy-procedure" object of the same shape. For upwards compatibility, procedure definitions with no lazy arguments are bound to ordinary lambdas.

```
; Changes to definition evluation
(define (definition-value exp)
  (cond ((symbol? (cadr exp)) (caddr exp))
        ((contains-lazy-args? (cadr exp))
         (make-lazy-lambda (arg-names (cdadr exp))
                           (arg-lazinesses (cdadr exp)))
                           (cddr exp)))
        (else (make-lambda (cdadr exp) (cddr exp)))))

(define (contains-lazy-args? exp)
  (cond ((null? exp) #f)
        ((pair? (car exp)) #t)
        (else (contains-lazy-args? (cdr exp)))))

(define (arg-names exp)
  (cond ((null? exp) '())
        ((pair? (car exp))
         (cons (caar exp) (arg-names (cdr exp))))
        (else (cons (car exp) (arg-names (cdr exp))))))

(define (arg-lazinesses exp)
  (cond ((null? exp) '())
        ((pair? (car exp))
         (if (memq (cadar exp) '(strict lazy lazy-memo))
             (cons (cadar exp) (arg-lazinesses (cdr exp)))
             (error ("Unrecognized laziness:" (cadar exp)))))
        (else (cons 'strict (arg-lazinesses (cdr exp)))))))

;; Lazy lambdas
(define (lazy-lambda? exp)
  (tagged-list? exp 'lazy-lambda))
;; As with lambda, the body is a list
(define (make-lazy-lambda vars lazinesses body)
  (cons 'lazy-lambda (cons vars (cons lazinesses body))))
(define (lazy-lambda-parameters exp) (cadr exp))
(define (lazy-lambda-lazinesses exp) (caddr exp))
(define (lazy-lambda-body exp) (cdddr exp))

;; Lazy procedures
;; procedure-body and procedure-environment are modified for the new case
(define (lazy-compound-procedure? procedure)
  (tagged-list? procedure 'lazy-procedure))
(define (make-lazy-procedure vars lazinesses body env)
  (list 'lazy-procedure vars lazinesses body env))
(define (procedure-parameters p) (cadr p))
(define (procedure-lazinesses p)
  (if (lazy-compound-procedure? p)
      (caddr p)
      (error "Attempted to lazily evaluate non-lazy procedure" p)))
(define (procedure-body p)
  (if (lazy-compound-procedure? p)
      (cadddr p)
      (caddr p)))
(define (procedure-environment p)
  (if (lazy-compound-procedure? p)
      (car (cddddr p))
      (cadddr p)))

;; Forcing and delaying
(define (actual-value exp env)
  (force-it (evaln exp env)))
(define (list-of-arg-values exps env)
  (if (no-operands? exps)
      '()
      (cons (actual-value (first-operand exps) env)
            (list-of-arg-values (rest-operands exps) env))))
(define (list-of-delayed-args exps lazinesses env)
  (if (no-operands? exps)
      '()
      (cons (maybe-delay-it (first-operand exps) (car lazinesses) env)
            (list-of-delayed-args (rest-operands exps) (cdr lazinesses) env))))
(define (maybe-delay-it exp laziness env)
  (cond ((eq? laziness 'strict) (actual-value exp env))
        ((eq? laziness 'lazy) (list 'thunk exp env))
        ((eq? laziness 'lazy-memo) (list 'memoizable-thunk exp env))
        (else (error "Unrecognized laziness for arg:" exp laziness))))
(define (memoizable-thunk? obj)
  (tagged-list? obj 'memoizable-thunk))
(define (force-it obj)
  (cond ((thunk? obj) (actual-value (thunk-exp obj) (thunk-env obj)))
        ((memoizable-thunk? obj)
         (let ((result (actual-value (thunk-exp obj) (thunk-env obj))))
           (set-car! obj 'evaluated-thunk))))
```

```

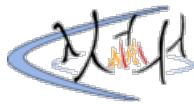
(set-car! (cdr obj) result)
(set-cdr! (cdr obj) '())
result))
((evaluated-thunk? obj) (thunk-value obj))
(else obj)))

;; Evaluator core
(define (evaln-lazy-optional exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if-lazy exp env)) ;; retained from lazy evaluator
        ((lambda? exp)
         (make-procedure (lambda-parameters exp) (lambda-body exp) env))
        ((lazy-lambda? exp)
         (make-lazy-procedure (lazy-lambda-parameters exp)
                               (lazy-lambda-lazinesses exp)
                               (lazy-lambda-body exp)
                               env)) ;; new
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (evaln (cond->if exp) env))
        ((and? exp) (evaln (and->if exp) env))
        ((or? exp) (evaln (or->if exp) env))
        ((not? exp) (evaln (not->if exp) env))
        ((let? exp) (evaln (let->combination exp) env))
        ((let*? exp) (evaln (let*->nested-lets exp) env))
        ((letrec? exp) (evaln (letrec->let exp) env))
        ((while? exp) (evaln (while->recursion exp) env))
        ((for? exp) (evaln (for->while exp) env))
        ((unless? exp) (evaln (unless->if exp) env))
        ((application? exp)
         (applyn (actual-value (operator exp) env)
                (operands exp)
                env))
        (else (error "Unknown expression type: EVAL" exp))))
)

(define (applyn-lazy-optional procedure arguments env)
  (cond ((primitive-procedure? procedure) (apply-primitive-procedure
                                                       procedure
                                                       (list-of-arg-values arguments env)))
        ((compound-procedure? procedure)
         (eval-sequence (procedure-body procedure)
                       (extend-environment (procedure-parameters procedure)
                                          (list-of-arg-values arguments env)
                                          (procedure-environment procedure))))
        ((lazy-compound-procedure? procedure)
         (eval-sequence (procedure-body procedure)
                       (extend-environment (procedure-parameters procedure)
                                          (list-of-delayed-args arguments
                                                               (procedure-lazinesses
                                                               procedure)
                                                               env)
                                          (procedure-environment procedure))))
        (else (error "Unknown procedure type: APPLYN" procedure))))
)

(define evaln evaln-lazy-optional)
(define applyn applyn-lazy-optional)

```



# sicp-ex-4.32

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.31) | Index | Next exercise (4.33) >>

meteorgan

In chapter 3, the `car` is **not** lazy. but here `car` and `cdr` are all lazy-evaluated. then we can build a lazy tree, all the branches of the tree are lazy-evaluated.

Sphinxsky

```
; different examples
; in the normal interpreter, "this is car" is printed
(define show-1
  (cons-stream
    (begin
      (display 'this-is-car)
      'this-is-car)
    (begin
      (display 'this-is-cdr)
      'this-is-cdr)))

; in the lazy evaluation interpreter, it is not
(define show-2
  (cons
    (begin
      (display 'this-is-car)
      'this-is-car)
    (begin
      (display 'this-is-cdr)
      'this-is-cdr)))

; examples of applications
; in lazy evaluators, you don't have to worry about the order of definitions

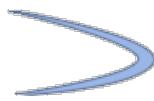
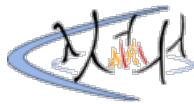
(define infinite-matrix-ones (cons ones infinite-matrix-ones))

(define ones (cons 1 ones))

(define (matrix-ref matrix x y) (list-ref (list-ref matrix x) y))

(matrix-ref infinite-matrix-ones 5011 10)
```

# sicp-ex-4.33



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.32) | Index | Next exercise (4.34) >>

meteorgan

`;; '(a b c) is equal to (quote (a b c)). so we should change the code in text-of-quotation like this.`

```
(define prev-eval eval)

(define (eval expr env)
  (if (quoted? expr)
      (text-of-quotation expr env)
      (prev-eval expr env)))

(define (quoted? expr) (tagged-list? expr 'quote))

(define (text-of-quotation expr env)
  (let ((text (cadr expr)))
    (if (pair? text)
        (evaln (make-list text) env)
        text)))

(define (make-list expr)
  (if (null? expr)
      (list 'quote '())
      (list 'quote (car expr))
      (make-list (cdr expr)))))
```

Felix021

`;; this version relies on the implementation of cons/car/cdr.`

```
(define (text-of-quotation expr)

  (define (new-list pair)
    (if (null? pair)
        '()
        (make-procedure
          ' (m)
          (list (list 'm 'car-value 'cdr-value))
          (extend-environment
            (list 'car-value 'cdr-value)
            (list (car pair) (new-list (cdr pair)))
            the-empty-environment)))

  (let ((text (cadr expr)))
    (if (not (pair? text))
        text
        (new-list text))))
```

atupal

Support nested list:

```
(define (text-of-quotation-lazy exp)
  (define (quotation->cons exp)
    (if (null? exp)
        '()
        (if (pair? exp)
            (list 'cons (quotation->cons (car exp)) (quotation->cons (cdr exp)))
            `',(exp)))

  (let ((env (cons (make-frame '() '() '())))
        (eval `(define (cons x y) (lambda (m) (m x y))) env)
        (eval `(define (car z) (z (lambda (p q) p))) env)
        (eval `(define (cdr z) (z (lambda (p q) q))) env)
        (let ((text (cadr exp))))
```

```

(if (pair? text)
  (eval (quotation->cons text) env)
  text)))
; if do so, all of the cons will be the lazy one even if they have been defined already
(define text-of-quotation text-of-quotation-lazy)

;(car '(a b c))
; output: 'a
;
;(car (car '((a b) (c (d e)))))
; output: a
;
;'()
; output: ()

```

revc

```

;;; Exercise 4.33

;;; redefined text-of-quotation will transform the exp into its corresponding
;;; expression of meta-circular evaluator, such as '(cons <quotation> (cons
;;; <quotation> (cons ...))) or '<quotation>, and then evaluate that expression.
;;;
(define (text-of-quotation exp)
  (if (pair? (cadr exp))
    (eval (text->lazy-conses-exp (cadr exp)) the-global-environment)
    (cadr exp)))

(define (text->lazy-conses-exp exp)
  (cond [(pair? exp)
         `'(cons ,(text->lazy-conses-exp (car exp)) ,(text->lazy-conses-exp (cdr exp)))]
        [else (list 'quote exp)]))

;;; add desired procedure to the-global-environment
(eval `(define (cons x y)
          (lambda (m) (m x y))) the-global-environment)

(eval `(define (car z)
          (z (lambda (p q) p))) the-global-environment)

(eval `(define (cdr z)
          (z (lambda (p q) q))) the-global-environment)

(eval `(define (list-ref items n)
          (if (= n 0)
              (car items)
              (list-ref (cdr items) (- n 1)))) the-global-environment)

(eval `(define (map proc items)
          (if (null? items)
              '()
              (cons (proc (car items))
                    (map proc (cdr items))))) the-global-environment)

(eval `(define (scale-list items factor)
          (map (lambda (x) (* x factor))
                items)) the-global-environment)

(eval `(define (add-lists list1 list2)
          (cond ((null? list1) list2)
                ((null? list2) list1)
                (else (cons (+ (car list1) (car list2))
                            (add-lists (cdr list1) (cdr list2)))))) the-global-
environment)

```

Sphinxsky

```
(define (quoted? exp-)
```

```

(tagged-list? exp- 'quote))

; Define the symbolic variables of L-Eval in the internal evaluation process
; It is used to distinguish it from MIT-Scheme forms such as <'a>
; The <'a> storage form of evaluation in MIT-Scheme is (list 'quote 'a)
(define (make-symbol symbol-)
  (cons 'quote symbol-))

; Constructing symbolic expression in MIT-Scheme
(define (make-scheme-quote exp-)
  (list 'quote exp-))

(define (is-list? exp-)
  (tagged-list? exp- 'list-))

(define (get-list-items exp-)
  (cdr exp-))

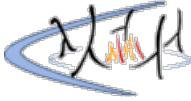
(define (make-cons exp-car exp-cdr)
  (list 'cons- exp-car exp-cdr))

(define (make-list items)
  (if (null? items)
    items
    (make-cons
      (car items)
      (make-list (cdr items)))))

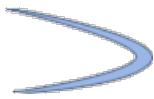
; Final evaluation of symbols
; L-Eval is implemented by using symbol data implementation in MIT-Scheme
(define (eval-quotation exp- env)
  (let ((text (cdr exp-)))
    (if (pair? text)
        ; symbolic data defined in MIT-Scheme
        (let ((data (car text)))
          (if (pair? data)
              ; if symbol list,converted to an lazy list evaluation
              (eval- (make-list (map make-scheme-quote data)) env)
              ; a single symbol is converted to an L-Eval internal evaluation
              expression
              (eval- (make-symbol data) env))
          ; a single symbol defined internally in l-eval is returned directly
          text)))
    (eval- exp- env)
    (cond ((self-evaluating? exp-) exp-)
          ((variable? exp-) (lookup-variable-value exp- env))
          ((quoted? exp-) (eval-quotation exp- env))
          ((is-list? exp-) (eval- (make-list (get-list-items exp-)) env))
          ((assignment? exp-) (eval-assignment exp- env))
          ((definition? exp-) (eval-definition exp- env))
          ((if? exp-) (eval-if exp- env))
          ((lambda? exp-)
           (make-procedure
             (lambda-parameters exp-)
             (lambda-body exp-)
             env))
          ((begin? exp-) (eval-sequence (begin-actions exp-) env))
          ((cond? exp-) (eval- (cond->if exp-) env))
          ((application? exp-)
           (apply-
             (actual-value (operator exp-) env)
             (operands exp-)
             env))
          (else (error "Unknown expression type -- EVAL" exp-))))
    ; => (car- (car- (cdr- (car- (cdr- (list- '(a b) '(c (list- d e))))))))
    ; => list-

```





# sicp-ex-4.34



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

[<< Previous exercise \(4.33\) | Index | Next exercise \(4.35\) >>](#)

Felix021

```
; ; based on 4-33

(map (lambda (name obj)
    (define-variable! name (list 'primitive obj) the-global-environment))
  (list 'raw-cons 'raw-car 'raw-cdr)
  (list cons car cdr))

(actual-value
 '(begin

  (define (cons x y)
    (raw-cons 'cons (lambda (m) (m x y)))))

  (define (car z)
    ((raw-cdr z) (lambda (p q) p)))

  (define (cdr z)
    ((raw-cdr z) (lambda (p q) q)))
)
the-global-environment)

(define (disp-cons obj depth)
(letrec ((user-car (lambda (z)
    (force-it (lookup-variable-value 'x (procedure-environment (cdr z))))))
        (user-cdr (lambda (z)
    (force-it (lookup-variable-value 'y (procedure-environment (cdr z)))))))
  (cond
    ((>= depth 10)
     (display "... "))
    ((null? obj)
     (display ""))
    (else
      (let ((cdr-value (user-cdr obj)))
        (display "(")
        (display (user-car obj))
        (if (tagged-list? cdr-value 'cons)
            (begin
              (display " ")
              (disp-cons cdr-value (+ depth 1)))
            (begin
              (display ". ")
              (display cdr-value)))
        (display ")")))))

(define (user-print object)
(if (compound-procedure? object)
  (display
    (list 'compound-procedure
      (procedure-parameters object)
      (procedure-body object)
      '<procedure-env>))
  (if (tagged-list? object 'cons)
    (disp-cons object 0)
    (display object)))))

(driver-loop)
```

awkravchuk

Arguably a bit more elegant solution:

```

                           the-empty-environment)))
(define-variable! 'true true initial-env)
(define-variable! 'false false initial-env)
(eval
  '(begin
    (define (cons cons-first cons-rest)
      (lambda (m) (m cons-first cons-rest)))
    (define (car z)
      (z (lambda (p q) p)))
    (define (cdr z)
      (z (lambda (p q) q)))
    (define (null? c)
      (equal? c '())))
  initial-env)
initial-env)

(define (lookup-variable-value* var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
              (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (car vals))
            (else (scan (cdr vars) (cdr vals))))))
    (if (eq? env the-empty-environment)
        '()
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))

(define (lazy-cons? procedure)
  (let ((env (procedure-environment procedure)))
    (and (not (null? (lookup-variable-value* 'cons-first env)))
         (not (null? (lookup-variable-value* 'cons-rest env))))))

(define (lazy-cons-print object)
  (define (lazy-cons-print-internal object n)
    (if (not (null? object))
        (let* ((env (procedure-environment object))
               (first (lookup-variable-value* 'cons-first env))
               (rest (lookup-variable-value* 'cons-rest env)))
          (if (> n 10)
              (display "...")
              (begin
                (let ((first-forced (force-it first)))
                  (if (and (compound-procedure? first-forced)
                            (lazy-cons? first-forced))
                      (lazy-cons-print first-forced)
                      (display first-forced)))
                (display " ")
                (lazy-cons-print-internal
                  (force-it rest)
                  (+ n 1))))))
        (display "(")
        (lazy-cons-print-internal object 0)
        (display ")")))
  (define (user-print object)
    (if (compound-procedure? object)
        (if (lazy-cons? object)
            (lazy-cons-print object)
            (display (list 'compound-procedure
                           (procedure-parameters object)
                           (procedure-body object)
                           '<procedure-env>)))
        (display object)))

;; To test:
;; '(a b (c d))
;; (define ones (cons 1 ones))
;; ones

```

imelendez

Just to pose the question, wouldn't this implementation regard any procedure with an environment containing 'cons-first and 'cons-rest as a cons'ed value, as illustrated below?

```

(define (make-fake-cons)
  (define cons-first 1)
  (define cons-rest 2)
  (lambda (m) 'fooled-you))

```

```
(define a-fake-cons (make-fake-cons))

(lazy-cons? a-fake-cons) ; true, even though it really isn't
```

Perhaps the interpreter can be meaningfully adapted to handle this situation.

davl

### Less nesting

```
(define (serialize-lazy-pairs object depth)
  (if (<= depth 9) ; THRESHOLD HERE
      (list 'lazy-pair
            (serialize-object (actual-value 'x (procedure-environment object)) (+ depth 1))
            '.
            (serialize-object (actual-value 'y (procedure-environment object)) (+ depth 1)))
      (list 'lazy-pair "...")))

(define (serialize-compound-procedure object)
  (if (compound-procedure? object)
      (list 'compound-procedure
            (procedure-parameters object)
            (procedure-body object)
            '<procedure-env>
            object)))

(define (lazy-pair? object)
  '*varies*)

(define (serialize-object object depth)
  (if (compound-procedure? object)
      (if (lazy-pair? object)
          (serialize-lazy-pairs object (+ 1 depth))
          (serialize-compound-procedure object))
      object))

(define (user-print object)
  (newline)
  (display (serialize-object object 0)))
```

revc

### Print cyclic or infinite list in some reasonable way — with tag replacements.

```
; Exercise 4.34

(define primitive-procedures
  (list (list 'car-in-underly-scheme car) ; preserved but renamed
        (list 'cdr-in-underly-scheme cdr) ; preserved but renamed
        (list 'cons-in-underly-scheme cons) ; preserved but renamed
        (list 'null? null?)
        (list 'list list)
        (list '+ +)
        (list '- -)
        (list '* *)
        (list '/ /)
        (list '= =)
        (list 'newline newline)
        (list 'display display)
  ; more primitives
  ))

(define the-global-environment (setup-environment))

;; represent pair of META-CIRCULAR as pair of SCHEME which is composed of a tag 'cons
;; and a lexical closure(i.e. a procedure of META-CIRCULAR).
(eval '(define (cons x y)
         (cons-in-underly-scheme 'pair? (lambda (m) (m x y)))) the-global-environment)

(eval '(define (car z)
         ((cdr-in-underly-scheme z) (lambda (p q) p))) the-global-environment)

(eval '(define (cdr z)
         ((cdr-in-underly-scheme z) (lambda (p q) q))) the-global-environment)
```

```

;; predicate that check if an object is a pair of META-CIRCULAR
(define (meta-pair? object)
  (tagged-list? object 'pair?))

;; print pair of META-CIRCULAR
(define (print-pair object)

  (define counter 0) ; the number of pairs which were revisited

  ; use a pair as key and the corresponding number of revisited times as value
  ; when the recursion to CAR and CDR of some pair ends, check if VALUE > 0,
  ; change VALUE to counter, and then increment counter by 1
  ; NOTE: We do not add things which are not pair into hashtable
  (define visited (make-hash-table))

  (define (put k v) (put-hash-table! visited k v)) ; an interface to visited for
  convenience
  (define (get k) (get-hash-table visited k #f)) ; an interface to visited for
  convenience
  (define (remove k) (remove-hash-table! visited k)) ; an interface to visited for
  convenience

  ; convert pair of META-CIRCULAR into pair of SCHEME for which we can handle more
  conveniently.
  ; The struct of converted pair is as follows:
  ;=====

  ; Printable-pair ::= ('be-referred-as Pair-of-META-CIRCULAR X Y))
  ; || ('refer-to Pair-of-META-CIRCULAR '*CAR* '*CDR*)
  ; || ('just-pair Pair-of-META-CIRCULAR X Y)
  ; || Not-pair

  ; X ::= #<Thunk CAR>
  ; || Printable-pair

  ; Y ::= #<Thunk CDR>
  ; || Printable-pair
  ;=====

  (define (convert-printable-pair object)

    ; if object is evaluated-thunk, then return its value, otherwise return a tag like #
    <Thunk {alternative}>
    (define (thunk-or-value object alternative)
      (if (evaluated-thunk? object)
          (convert-printable-pair (thunk-value object))
          (string-append "#<thunk " alternative ">"))

    ; we visit some pair in visited table again, so change the its value to the desired
    string
    (define (visit-again! object)
      (put object (+ 1 (get object)))
      (list 'refer-to object '*CAR* '*CDR*))

    (if (meta-pair? object)
        (cond [(get object) (visit-again! object)] ; visit again! return a string
              like "#{counter}"
              [else
               (put object 0) ; the first visit!
               (let ([x (thunk-or-value (eval 'x (procedure-environment (cdr object))) "CAR")]
                     [y (thunk-or-value (eval 'y (procedure-environment (cdr object))) "CDR")])
                 (if (zero? (get object)) ; no inner elements refer
                     to it
                     (begin
                       (remove object)
                       (list 'just-pair object x y)))
                     (begin
                       (put object counter)
                       (set! counter (+ counter 1)) ; increment counter
                       (list 'be-referred-as object x y))))]) ; return the processed
        pair
        object)) ; not pair, return directly

    (define (be-referred? object)
      (tagged-list? object 'be-referred-as))

    (define (referer? object)
      (tagged-list? object 'refer-to))

    ; print printable pair
    ; We need consider three cases:
  )

```

```

;; 1. a pair refers to its outer list, which will be printed as #{counter} where counter
is the corresponding
;; serial number of its outer list.
;; 2. a pair is referenced by its inner elements, which will be printed as #{counter}=(X
. Y) where counter is
;; its serial number
;; 3. a normal pair which does not refer to another pair and is not referenced by
others, that will be printed
;; as (X . Y)
;; NOTE: if Y is a pair, then print-pair won't print the preceding ". " and the
parentheses enclosing it.

(define (print-pair pair with-paren)
  (let* ([left (if with-paren "(" "")]
         [right (if with-paren ")" "")])
    [val (get (cadr pair))])
  [x (list-ref pair 2)]
  [y (list-ref pair 3)])
  [tag (if val (string-append "#"
    (number->string val))) val])
  [middle (if (null? y) "" " ")])]

(cond [(be-referred? pair) (display tag) (display "=") (display left)]
      [(referer? pair) (display tag) (display "#")]
      [else (display left)])]

(cond [(not (referer? pair))
       (cond [(pair? x) (print-pair x #t)]
             [else (display x)]))

       (display middle)

       (cond [(be-referred? y) (print-pair y #t)]
             [(referer? y) (display ". ")
              (print-pair y #f)]
             [(pair? y) (print-pair y #f)]
             [(null? y) (display "")]
             [else y (display ". ")
              (display y)])]

       (display right)])
  )))

(print-pair (convert-printable-pair object) #t))

(define (user-print object)
  (cond [(meta-pair? object) (print-pair object)] ; the clause for pair of META-CIRCULAR
        [(compound-procedure? object)
         (display (list 'compound-procedure
                       (procedure-parameters object)
                       (procedure-body object)
                       '<procedure-env>))]
        [else (display object)]))

;; Exercise 4.34 additional procedures
(eval '(define (list-ref items n)
         (if (= n 0)
             (car items)
             (list-ref (cdr items) (- n 1)))) the-global-environment)

(eval '(define (map proc items)
         (if (null? items)
             '()
             (cons (proc (car items))
                   (map proc (cdr items))))) the-global-environment)

(eval '(define (scale-list items factor)
         (map (lambda (x) (* x factor))
               items)) the-global-environment)

(eval '(define (add-lists list1 list2)
         (cond ((null? list1) list2)
               ((null? list2) list1)
               (else (cons (+ (car list1) (car list2))
                           (add-lists (cdr list1) (cdr list2)))))) the-global-
environment)

(eval '(define ones (cons 1 ones)) the-global-environment)

(eval '(define integers (cons 1 (add-lists ones integers))) the-global-environment)

(eval '(define (for-each proc items)
         (if (null? items)
             'done
             (begin (proc (car items))
                   (for-each proc (cdr items)))))) the-global-environment)

;;;;;;;;;;;;;;
;;;;;; test ;;;;;;;

```

```
;;;;;;;
;;;; ones ;;;;
;;;;;;;

;;; L-Eval input:
ones

;;; L-Eval value:
(#<thunk CAR> . #<thunk CDR>)

;;; L-Eval input:
(car ones)

;;; L-Eval value:
1

;;; L-Eval input:
ones

;;; L-Eval value:
(1 . #<thunk CDR>)

;;; L-Eval input:
(cdr ones)

;;; L-Eval value:
#0=(1 . #0#)

;;;;;;
;;;; integers ;;;;
;;;;;;;

;;; L-Eval input:
(list-ref integers 3)

;;; L-Eval value:
4

;;; L-Eval input:
integers

;;; L-Eval value:
(1 2 3 4 . #<thunk CDR>)

;;; L-Eval input:
(list-ref integers 20)

;;; L-Eval value:
21

;;; L-Eval input:
integers

;;; L-Eval value:
(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 . #<thunk CDR>)

;;;;;;
;;;; not cyclic but referenced ;;;;
;;;;;;;

;;; L-Eval input:
(define s (cons 1 2))

;;; L-Eval value:
ok

;;; L-Eval input:
(define w (cons s s))

;;; L-Eval value:
ok

;;; L-Eval input:
(car s)

;;; L-Eval value:
1

;;; L-Eval input:
(cdr s)

;;; L-Eval value:
2

;;; L-Eval input:
```

```

(car w)

;;; L-Eval value:
(1 . 2)

;;; L-Eval input:
(cdr w)

;;; L-Eval value:
(1 . 2)

;;; L-Eval input:
w

;;; L-Eval value:
((1 . 2) 1 . 2)

;;;;;;;;;;;;;;
;;;; list: special pair ;;;
;;;;;;;;;;;;;

;;; L-Eval input:
(define lst '(1 2 3 4))

;;; L-Eval value:
ok

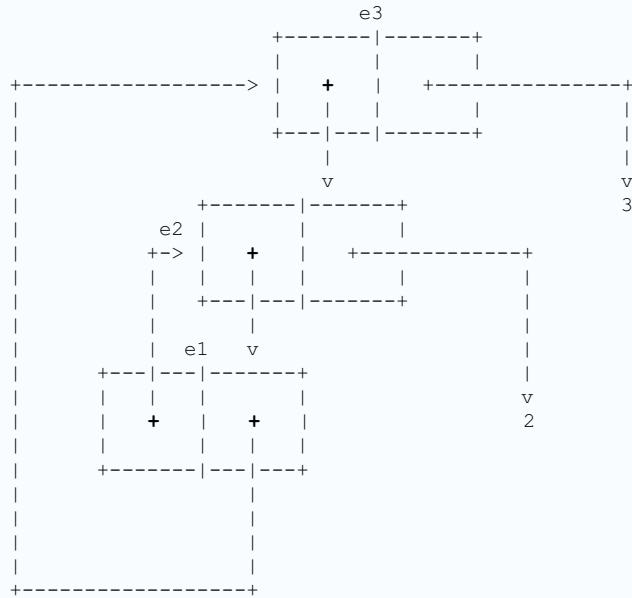
;;; L-Eval input:
(for-each (lambda (x) (display x)) lst)
1234
;;; L-Eval value:
done

;;; L-Eval input:
lst

;;; L-Eval value:
(1 2 3 4)

;;;;;;;;;;;;;;
;;;; Multiplex cycle ;;;
;;;;;;;;;;;;;

```



```

;;; L-Eval input:
(define c1 (cons 1 1))

;;; L-Eval value:
ok

;;; L-Eval input:
(define c2 (cons c1 2))

;;; L-Eval value:
ok

;;; L-Eval input:
(define c3 (cons c2 3))

;;; L-Eval value:
ok

```

```

;;; L-Eval input:
(define c1 (cons c2 c3))

;;;;;
Skip the access section
;;;;;

;;; L-Eval input:
c1

;;; L-Eval value:
#0=((#0# . 2) (#0# . 2) . 3)

;;; L-Eval input:
c3

;;; L-Eval value:
#1=(#0=((#0# . #1#) . 2) . 3)

```

Sphinxsky

a simpler implementation it is modified on the basis of 4.33

```

; add primitive procedure identity
(define primitive-procedures
  (list
    (list 'null?- null?)
    (list 'display- display)
    (list 'newline- newline)
    (list '= =)
    (list '/ /)
    (list '- -)
    (list '* *)
    (list '+ +)
    (list 'identity (lambda (x) x)))))

; define list expression
; =====
(define (is-list? exp)
  (tagged-list? exp- 'list-))

(define (is-empty-list? exp)
  (null? exp-))

(define (get-list-items exp)
  (cdr exp-))

(define (make-cons exp-car exp-cdr)
  (list 'cons- exp-car exp-cdr))

(define (make-list items)
  (if (null? items)
      items
      (make-cons
        (car items)
        (make-list (cdr items)))))

; =====

; add list operation
; =====
(define list-operation-exp-1
  '(define- (cons- x y)
      (define- x-Evaluated? false-)
      (define- y-Evaluated? false-)
      (define- x-Evaluated '())
      (define- y-Evaluated '())
      (lambda- (_cons-signal__)
        (cond- ((= _cons-signal_ 0)
                (if- x-Evaluated?
                    x-Evaluated
                    (begin-
                      (set!- x-Evaluated (identity x))
                      (set!- x-Evaluated? true-)
                      x-Evaluated)))
               ((= _cons-signal_ 1)
                (if- y-Evaluated?

```

```

y-Evaluated
(begin-
  (set!- y-Evaluated (identity y))
  (set!- y-Evaluated? true-)
  y-Evaluated)))
((= __cons-signal__ 2)
(if- x-Evaluated?
  x-Evaluated
  'promise-head))
((= __cons-signal__ 3)
(if- y-Evaluated?
  y-Evaluated
  'promise-tail)))))

(define list-operation-exp-2
  '(define- (car- z) (z 0)))

(define list-operation-exp-3
  '(define- (cdr- z) (z 1)))

(define list-operation-exp-4
  '(define- (list-ref- items n)
    (if- (= n 0)
      (car- items)
      (list-ref- (cdr- items) (- n 1)))))

(define list-operation-exp-5
  '(define- (map- proc items)
    (if- (null?- items)
      items
      (cons-
        (proc (car- items))
        (map- proc (cdr- items))))))

(define list-operation-exp-6
  '(define- (scale-list items factor)
    (map-
      (lambda- (x) (* x factor))
      items)))

(define list-operation-exp-7
  '(define- (add-lists list1 list2)
    (cond- ((null?- list1) list2)
      ((null?- list2) list1)
      (else-
        (cons-
          (+ (car- list1) (car- list2))
          (add-lists (cdr- list1) (cdr- list2)))))))

(define list-operation-exp-8
  '(define- (matrix-ref matrix x y)
    (list-ref- (list-ref- matrix x) y)))

(define list-operation-exp-9
  '(define- (append- x y)
    (if- (null?- x)
      y
      (cons- (car- x)
        (append- (cdr- x) y)))))

; import interpreter
(for-each
  (lambda- (exp-)
    (actual-value exp- the-global-environment))
(list
  list-operation-exp-1
  list-operation-exp-2
  list-operation-exp-3
  list-operation-exp-4
  list-operation-exp-5
  list-operation-exp-6
  list-operation-exp-7
  list-operation-exp-8
  list-operation-exp-9))
; =====

; predicate to add an empty list
(define (eval- exp- env)
  (cond ((self-evaluating? exp-) exp-)
    ((variable? exp-) (lookup-variable-value exp- env))
    ((quoted? exp-) (eval-quotation exp- env))
    ((is-empty-list? exp-) '())
    ((is-list? exp-) (eval- (make-list (get-list-items exp-)) env))
    ((assignment? exp-) (eval-assignment exp- env))
    ((definition? exp-) (eval-definition exp- env)))

```

```

((if? exp-) (eval-if exp- env))
((lambda? exp-)
  (make-procedure
    (lambda-parameters exp-)
    (lambda-body exp-)
    env))
((begin? exp-) (eval-sequence (begin-actions exp-) env))
((cond? exp-) (eval- (cond->if exp-) env))
((application? exp-)
  (apply-
    (actual-value (operator exp-) env)
    (operands exp-)
    env))
(else (error "Unknown expression type -- EVAL" exp-)))

; printing result processing
; =====
(define (make-compound-procedure-show cp)
  (list
    'compound-procedure
    (procedure-parameters cp)
    (procedure-body cp)
    '<procedure-env>))

(define (is-cons? object)
  (if (compound-procedure? object)
    (let ((parameters (procedure-parameters object)))
      (equal? parameters (list '__cons-signal__)))
    false))

(define (make-cons-show c) c)

(define (make-show object)
  (cond ((is-cons? object)
         (make-cons-show object))
        ((compound-procedure? object)
         (make-compound-procedure-show object))
        (else object)))

(define (make-cons-show c)
  (let* ((cons-env (procedure-environment c))
         (cons-head (apply- c (list 2) cons-env))
         (cons-tail (apply- c (list 3) cons-env)))
    (list
      (make-show cons-head)
      (make-show cons-tail)))))

(define (user-print object)
  (display (make-show object)))
; =====

```

closeparen

Interact between the interpreter and user-land to use the car and cdr procedures normally. It took a lot of banging my head against the wall to figure out the "quote" parts!

```

; user-print definition. driver-loop needs to be modified to pass env.

(define (user-print object env)
  (if (compound-procedure? object)
    (if (eq? (car (procedure-parameters object)) 'user-cons-arg)
      (display (actual-value (list 'take (list 'quote object) 2) env))
      (display (list 'compound-procedure
                    (procedure-parameters object)
                    (procedure-body object)
                    '<procedure-env>)))
    (display object)))

; test client

(define the-global-environment (setup-environment))
(driver-loop)

(define (take x n)
  (cond ((null? x) '())
        ((= n 0) (list "..."))
        (else (old-cons (car x) (take (cdr x) (- n 1))))))

(define (cons x y)
  (lambda (user-cons-arg) (user-cons-arg x y)))

```

```

(define (car z)
  (z (lambda (p q) p)))

(define (cdr z)
  (z (lambda (p q) q)))

(define ll (cons 'a (cons 'b (cons 'c '()))))

(car ll)
(car (cdr ll))
(car (cdr (cdr ll)))
ll ;; "(a b ...)"

```

SteeleDynamics

It took a while to figure out how to neatly encapsulate the mutual recursion required to print 'cons' pairs in the REPL. The unit tests were omitted for clarity.

1. Completed exercise 4.33.
2. Added 'cons', 'car', and 'cdr' procedures to environment 'E0'.
3. Defined 'max-length' and 'max-depth' to prevent infinite recursion.
4. Created 'cons?' predicate procedure that is almost alpha-equivalence (no pattern-matching/unification).
5. Noting that printing from REPL is a primitive procedure, displayed list elements are forced via 'force-it' procedure.
6. All printing is encapsulated within the two mutually recursive procedures 'lazy-print' and 'cons-print'.
7. Mutual recursion starts in 'user-print' procedure.

```

; starting with a completed exercise 4.33 implementation...

; setup environment E0 for lazy evaluator
(define E0 (setup-environment))

; (lazy-evaluator) cons constructor procedure
(actual-value '(define (cons a d) (lambda (m) (m a d))) E0)

; (lazy-evaluator) car selector procedure
(actual-value '(define (car z) (z (lambda (a d) a))) E0)

; (lazy-evaluator) cdr selector procedure
(actual-value '(define (cdr z) (z (lambda (a d) d))) E0)

; max-length definition ;!
(define max-length 8)

; max-depth definition ;!
(define max-depth 4)

; cons? predicate procedure ;!
; Or, "A 'cons' by any other name would construct as neat."
(define (cons? object)
  (and (compound-procedure? object)
       (equal? (procedure-parameters object) '(m))
       (equal? (procedure-body object) '((m a d)))))

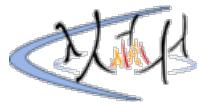
; cons-print procedure ;!
(define (cons-print object length depth)
  (let ((env (procedure-environment object)))
    (let ((a (force-it (lookup-variable-value 'a env)))
          (d (force-it (lookup-variable-value 'd env))))
      (if (= length max-length)
          (display "(")
          (display " "))
      (cond ((zero? length) (display "..."))
            ((zero? depth) (display "..."))
            ((cons? a) (lazy-print a max-length (- depth 1)))
            (else (lazy-print a length depth)))
      (cond ((zero? length) (display ")"))
            ((zero? depth) (display ")"))
            ((null? d) (display ")"))
            ((cons? d) (lazy-print d (- length 1) depth))
            (else (display ".")
                  (lazy-print d (- length 1) depth)
                  (display ")")))))

; lazy-print procedure ;!
(define (lazy-print object length depth)
  (cond ((cons? object)
         (cons-print object length depth))
        ((compound-procedure? object)
         (display (list 'compound-procedure
                       (procedure-parameters object)
                       (procedure-body object)
                       '<procedure-env>)))
        (else (display object))))
```

```
; user-print procedure ;!
(define (user-print object) (lazy-print object max-length max-depth))
```

---

Last modified : 2022-09-30 19:09:57  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-4.35

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

[<< Previous exercise \(4.34\)](#) | [Index](#) | [Next exercise \(4.36\) >>](#)

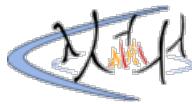
---

meteorgan

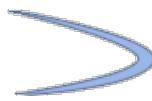
```
(define (an-interger-between low high)
  (require (<= low high))
  (amb low (an-interger-between (+ low 1) high)))
```

---

Last modified : 2012-08-02 23:08:08  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-4.36



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.35) | Index | Next exercise (4.37) >>

x davidiu

Here's an efficient method that iterates only over two of the numbers, not all three. We are able to do this here because of the lack of limitations that were present in exercise 4.35 (requiring  $k \leq high$ ) and exercise 3..69 (arbitrary input streams S, T, and U rather than just "all integers").

First, some helper functions:

```
(define (hypotenuse-squared i j)
  (+ (square i) (square j)))

(define (round-to-integer x)
  (inexact->exact (round x)))

(define (perfect-square? i)
  (= i (square (round-to-integer (sqrt i)))))
```

Next, a function that finds all pythagorean triples with  $i \leq j = \text{middle}$ , where middle is an argument.

```
(define (a-pythagorean-triple-with-middle-fixed middle)
  (let ((i (an-integer-between 1 middle)))
    (let ((hypot2 (hypotenuse-squared i middle)))
      (require (perfect-square? hypot2))
      (list i middle (sqrt hypot2)))))
```

Finally, a function that generates all triples:

```
(define (all-pythagorean-triples)
  (let ((middle (an-integer-starting-from 1)))
    (let ((triple (a-pythagorean-triple-with-middle-fixed middle)))
      triple)))
```

meteorgan

```
(define (a-pythagorean-triple-greater-than low)
  (let ((k (an-integer-starting-from low)))
    (let ((i (an-integer-between low k)))
      (let ((j (an-integer-between i k)))
        (require (= (+ (* i i) (* j j)) (* k k)))
        (list i j k)))))
```

revc

an improved solution with a uniform style from meteorgan

```
(define (a-pythagorean-triple-greater-than low)
  (let ((i (an-integer-starting-from low)))
    (let ((j (an-integer-between low i)))
      (let ((k (an-integer-between low j)))
        (require (= (+ (* k k) (* j j)) (* i i)))
        (list k j i)))))
```

squeegie

:shorter, but very inefficient!

```
(define (a-pythagorean-triple-greater-than low)
  (let ((high (an-integer-starting-from low)))
    (a-pythagorean-triple-between 1 higher)))
```

Rptx

```
; using a helper procedure to generate integers smaller than a certain number

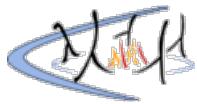
(define (an-integer-to n)
  (require (> n 0))
  (amb n (an-integer-to (- n 1)))))

(define (a-pythagorean-triple)
  (let ((k (an-integer-from 1)))
    (let ((j (an-integer-to k)))
      (let ((i (an-integer-to j)))
        (require (= (+ (* i i)
                      (* j j))
                    (* k k)))
        (list i j k)))))
```

Anon

This solution uses the triangle inequality (i.e. the sum of the lengths of any two sides is greater than the third length) to bound the potential values of k (along with the fact that  $k > j$  since it is the hypotenuse).

```
(define (a-pythagorean-triple)
  (let ((j (an-integer-starting-from 1)))
    (let ((i (an-integer-between 1 j)))
      (let ((k (an-integer-between (+ j 1) (+ i j -1))))
        (require (= (+ (* i i) (* j j)) (* k k)))
        (list i j k)))))
```



# sicp-ex-4.37

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

<< Previous exercise (4.36) | Index | Next exercise (4.38) >>

---

meteorgan

yes. Ben's solution is more efficient. It ignored many irrelevant solutions. because it doesn't need to enumerate k from j to high.

uuu

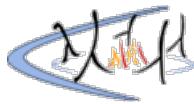
I think it's different.

When the range from *low* to *high* is not large, ex4.35 method will work faster, because *sqrt* and *integer?* are not fast.

When the range from *low* to *high* is large, ex4.37 method will work faster, because it ignored many irrelevant solutions.

---

Last modified : 2017-01-02 14:31:37  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-4.38

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.37) | Index | Next exercise (4.39) >>

meteorgan

```
;;; Starting a new problem
;;; Amb-Eval output:
((baker 1) (cooper 2) (fletcher 4) (miller 3) (smith 5))

;;; Amb-Eval input:
try-again

;;; Amb-Eval output:
((baker 1) (cooper 2) (fletcher 4) (miller 5) (smith 3))

;;; Amb-Eval input:
try-again

;;; Amb-Eval output:
((baker 1) (cooper 4) (fletcher 2) (miller 5) (smith 3))

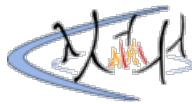
;;; Amb-Eval input:
try-again

;;; Amb-Eval output:
((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))

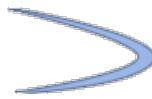
;;; Amb-Eval input:
try-again

;;; Amb-Eval output:
((baker 3) (cooper 4) (fletcher 2) (miller 5) (smith 1))
```

Last modified : 2021-12-22 01:18:49  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-4.39



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.38) | Index | Next exercise (4.40) >>

meteorgan

The order of restrictions doesn't affect the result. but actually affect the running time.  
for example this code is more efficient:

```
(define (mutiple-dwelling)
  (let ((baker (amb 1 2 3 4 5))
        (cooper (amb 1 2 3 4 5))
        (fletcher (amb 1 2 3 4 5)))
    (miller (amb 1 2 3 4 5))
    (smith (amb 1 2 3 4 5)))
  (require (not (= baker 5)))
  (require (not (= cooper 1)))
  (require (not (= fletcher 5)))
  (require (not (= fletcher 1)))
  (require (> miller cooper))
  (require (not (= (abs (- smith fletcher)) 1)))
  (require (not (= (abs (- fletcher cooper)) 1)))
  (require (distinct? (list baker cooper fletcher miller smith)))
  (list (list 'baker baker)
        (list 'cooper cooper)
        (list 'fletcher fletcher)
        (list 'miller miller)
        (list 'smith smith)))
```

Because distinct? runs in quadratic time, while other conditions can be assured in constant time. when moved it to the end of restrictions, it runs less times than before. so reduce the total time.

x davidiu

Note that this exercise can be solved purely using some elementary combinatorial calculations on pencil and paper, without running the code or even knowing how long it takes to run the distinct? procedure.

The first part, whether the ordering of requirements affects the final result, is clearly "no".

To answer the second part, let's define four "events", corresponding to the four require statements after the initial distinct? requirement:

1. baker = 5
2. cooper = 1
3. fletcher = 5
4. fletcher = 1

Note that the number of distinct outcomes is  $5! = 120$ . Of those 120, the number that fail the first requirement, e.g. match event 1, we denote as  $N(1) = 4! = 24$ . The number that fail \*either\* the first or second requirement, e.g. match \*either\* event 1 or event 2, we denote as  $N(1 \text{ or } 2)$ . Using "the principle of inclusion and exclusion" (c.f. wikipedia), we have

$$N(1 \text{ or } 2) = N(1) + N(2) - N(1 \text{ and } 2) = 2 * 4! - 3! = 42$$

An easy way to think about this is we are adding up overlapping Venn diagram regions and correcting for double-counting overlaps.

Continuing, we have after some further calculation:

$$\begin{aligned} N(1 \text{ or } 2 \text{ or } 3) &= 3 * 4! - 2 * 3! = 60 \\ N(1 \text{ or } 2 \text{ or } 3 \text{ or } 4) &= 4 * 4! - 4 * 3! = 72 \end{aligned}$$

From these numbers we see that 24 outcomes fail the first requirement, 18 fail the second, 18 fail the third, and 12 fail the fourth. Hence, the total number of tests done on these failed requirements, during the four requirements examined here, is

$$1 * 24 + 2 * 18 + 3 * 18 + 4 * 12 = 162.$$

Now let's see what happens if we reorder the four events, which correspond to the four require statements in the code after the initial distinct? require statement:

```

1. fletcher = 5
2. fletcher = 1
3. baker = 5
4. cooper = 1

```

Performing a similar calculation as above (which I will leave as an exercise for the reader), we obtain the total number of tests done during the four requirements is 156.

Hence, assuming that our Scheme interpreter takes the same time to lookup each of the variables in the environment, having the two fletcher requirements \*before\* the baker and cooper requirements is \*guaranteed\* to be faster.

Note that we are able to compare these two orderings because everything that happens after the four requirements is the same for both.

revc

If the restrictions were set, then the order of them does not matter to the answer.

As far as I'm concerned that the solution provided by meteorgan is not the most efficient one. However, meteorgan was right about one thing that we should move distinct? to the end of restrictions.

To Improve efficiency, one thing we need to know is how do multiple ambs search. I will give a simple example to illustrate:

```

;;; Amb-Eval input:
(define (t) (list (amb 1 2) (amb 'a 'b)))

;;; Starting a new problem
;;; Amb-Eval value:
ok

;;; Amb-Eval input:
(t)

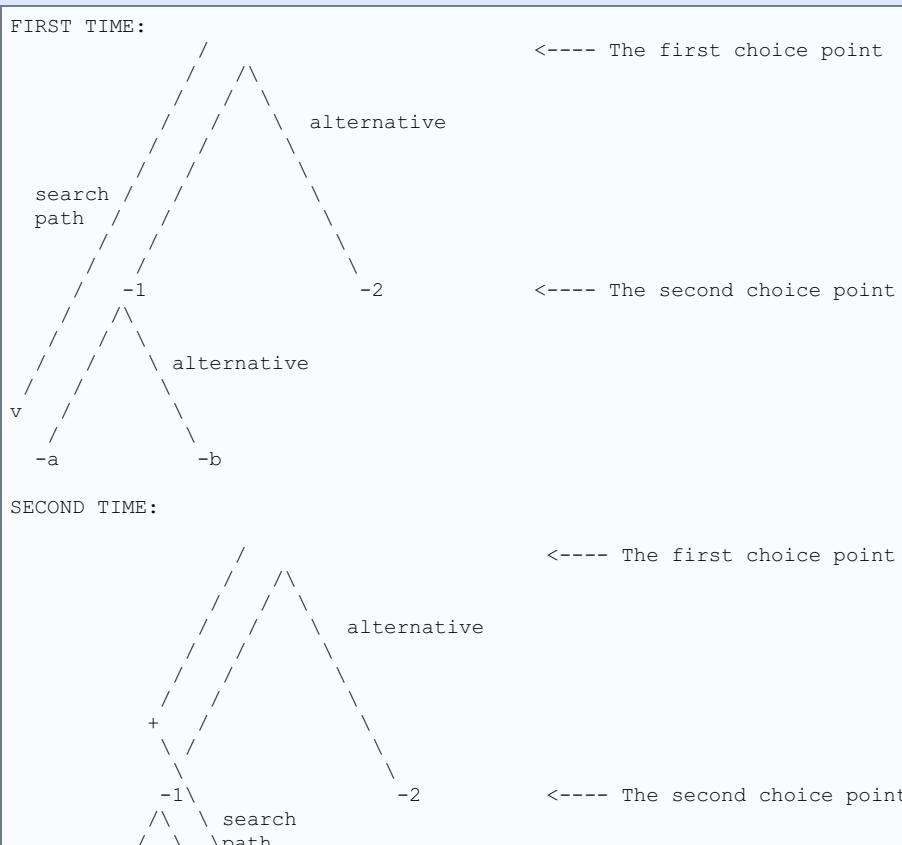
;;; Starting a new problem
;;; Amb-Eval value:
(1 a)

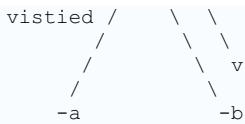
;;; Amb-Eval input:
try-again

;;; Amb-Eval value:
(1 b)

```

The following timing diagram depicts the search process above:





This is analogous to nested loop which try to run through the inner loop before picking up the next element of the outer loop. In that case, we take 1 as the first choice, and then explore all available choices. After finishing all attempts, we will go back to the first choice point, and select the alternative 2, and then perform a further search.

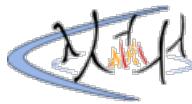
To avoid unnecessary searches, we need avoid making the wrong choice as soon as possible. If we find the early choice is wrong, it's too late to stop it, which means we must run through all further choices, even all of them are needless. So, all I do is try to find wrong choices early and stop them in time.

My solution is as follows:

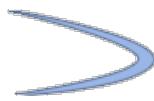
```

(define (multiple-dwelling)
  (let ([fletcher (amb 1 2 3 4 5)])
    (require (not (= fletcher 1)))
    (require (not (= fletcher 5)))
    (let ([cooper (amb 1 2 3 4 5)])
      (require (not (= cooper 1)))
      (require (not (= (abs (- fletcher cooper)) 1)))
      (let ([smith (amb 1 2 3 4 5)])
        (require (not (= (abs (- smith fletcher)) 1)))
        (let ([miller (amb 1 2 3 4 5)])
          (require (> miller cooper))
          (let ([baker (amb 1 2 3 4 5)])
            (require (not (= baker 5)))
            (require
              (distinct? (list baker cooper fletcher miller smith)))
            (list (list 'baker baker)
                  (list 'cooper cooper)
                  (list 'fletcher fletcher)
                  (list 'miller miller)
                  (list 'smith smith))))))))

```



# sicp-ex-4.40



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.39) | Index | Next exercise (4.41) >>

meteorgan

```
(define (multiple-dwelling)
  (let ((cooper (amb 2 3 4 5))
        (miller (amb 3 4 5)))
    (require (> miller cooper))
    (let ((fletcher (amb 2 3 4)))
      (require (not (= (abs (- fletcher cooper)) 1)))
      (let ((smith (amb 1 2 3 4 5)))
        (require (not (= (abs (- smith fletcher)) 1)))
        (let ((baker (amb 1 2 3 4)))
          (require (distinct? (list baker cooper fletcher miller smith)))
          (list (list 'baker baker)
                (list 'cooper cooper)
                (list 'fletcher fletcher)
                (list 'miller miller)
                (list 'smith smith)))))))
```

Shaw

```
(define (multiple-dwelling)
  (let ((cooper (amb 2 3 4 5))
        (miller (amb 3 4 5)))
    (require (> miller cooper))
    (let ((fletcher (amb 2 3 4)))
      (require (not (or (= fletcher cooper)
                      (= fletcher miller)
                      (= (abs (- cooper fletcher)) 1))))
      (let ((smith (amb 1 2 3 4 5)))
        (require (not (or (= smith cooper)
                      (= smith miller)
                      (= smith fletcher)
                      (= (abs (- smith fletcher)) 1))))
        (let ((baker (amb 1 2 3 4)))
          (require (distinct? (list baker cooper fletcher smith miller)))
          (list (list 'baker baker)
                (list 'cooper cooper)
                (list 'fletcher fletcher)
                (list 'miller miller)
                (list 'smith smith)))))))
```

felix021

The use of 'an-integer-between' from 4-35 helps a lot here.

```
(define (multiple-dwelling)
  (let* ((baker (amb 1 2 3 4))
        (cooper (amb 2 3 4 5))
        (fletcher (amb 2 3 4)))
    (require (not (= (abs (- fletcher cooper)) 1)))
    (let* ((miller (an-integer-between (+ 1 cooper) 5))
           (smith (amb
                   (an-integer-between 1 (- fletcher 2))
                   (an-integer-between (+ fletcher 2) 5))))
      (require
       (distinct? (list baker cooper fletcher miller smith)))
      (list
       (list 'baker baker)
       (list 'cooper cooper)
       (list 'fletcher fletcher)
```

```
(list 'miller miller)
(list 'smith smith))))
```

x davidiu

I agree with SophiaG below: the exercise asks for a \*nondeterministic\* program, which means that you can only rule out possibilities using require, and not manually exclude them, even if they are trivial. (If you were allowed to manually exclude outcomes, there would be nothing stopping us from writing a trivial function that just directly outputs the single answer).

Anyways, here's my solution (there's no need to even use distinct? at all, since it's significantly more efficient, at the cost of a small number of extra lines of code, to just manually type out the equivalent require statements:

```
(define (multiple-dwelling)
  (let ((fletcher (amb 1 2 3 4 5)))
    (require (not (= fletcher 5)))
    (require (not (= fletcher 1)))
    (let ((smith (amb 1 2 3 4 5)))
      (require (not (= smith fletcher)))
      (require (not (= 1 (abs (- smith fletcher))))))
      (let ((cooper (amb 1 2 3 4 5)))
        (require (not (= cooper 1)))
        (require (not (= cooper smith)))
        (require (not (= cooper fletcher)))
        (require (not (= 1 (abs (- fletcher cooper))))))
        (let ((miller (amb 1 2 3 4 5)))
          (require (> miller cooper))
          (require (not (= miller fletcher)))
          (require (not (= miller smith)))
          (let ((baker (amb 1 2 3 4 5)))
            (require (not (= baker 5)))
            (require (not (= baker cooper)))
            (require (not (= baker fletcher)))
            (require (not (= baker miller)))
            (require (not (= baker smith)))
            (list (list 'baker baker)
                  (list 'cooper cooper)
                  (list 'fletcher fletcher)
                  (list 'miller miller)
                  (list 'smith smith))))))))
```

SophiaG

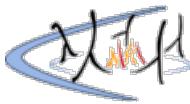
I'm not sure about these solutions. If you chop down the set of assignments to eliminate some of the requires you're: a) solving some of the problem for the program, and b) eliminating some of the search tree for later requires. For example, I would argue the requires with relations between two people should have full floor lists for both if one of them cannot occupy all floors. Otherwise, your solution may still work, but may not in modified cases such as Exercise 4.38. Given that interpretation, here's the best optimization I could come up with:

```
(define (nested-multiple-dwelling)
  (let ((fletcher (amb 1 2 3 4 5)))
    (require (not (= fletcher 5)))
    (require (not (= fletcher 1)))
    (let ((cooper (amb 1 2 3 4 5)))
      (baker (amb 1 2 3 4 5)))
    (require (not (= baker 5)))
    (require (not (= cooper 1)))
    (let ((miller (amb 1 2 3 4 5)))
      (require (> miller cooper))
      (let ((smith (amb 1 2 3 4 5)))
        (require (not (= (abs (- fletcher cooper)) 1)))
        (require (not (= (abs (- smith fletcher)) 1)))
        (require
          (distinct? (list baker cooper fletcher miller smith)))
        (list (list 'baker baker)
              (list 'cooper cooper)
              (list 'fletcher fletcher)
              (list 'miller miller)
              (list 'smith smith))))))))
```

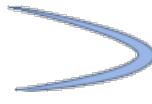
ce

Although I agree with your reasoning about not manually ruling combinations out, SofiaG, I don't think your solution actually changes the runtime of the program in any meaningful way: even though you've inserted a bunch of lets, the calls to amb would still be happening in the same order as in the original program from the text. I think the secret sauce is actually adding more requires, perhaps by calling distinct after every time that we define a new dweller, for example.





# sicp-ex-4.41



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.39) | Index | Next exercise (4.42) >>

woofy

Backtracking with plain recursion. No permutations.

```
(define baker 0)
(define cooper 1)
(define fletcher 2)
(define miller 3)
(define smith 4)

(define (list-ref s n)
  (cond ((null? s) false)
        ((= n 0) (car s))
        (else (list-ref (cdr s) (- n 1)))))

(define (floor who partial head)
  (list-ref partial (- head who)))

(define (check set who)
  (let ((baker-floor (floor baker set who))
        (cooper-floor (floor cooper set who))
        (fletcher-floor (floor fletcher set who))
        (miller-floor (floor miller set who))
        (smith-floor (floor smith set who)))
    ;(display (list "checking..." set who))
    (cond ((= who baker) (not (= baker-floor 5)))
          ((= who cooper)
           (and (not (= cooper-floor baker-floor))
                (not (= cooper-floor 1))))
          ((= who fletcher)
           (and (not (= fletcher-floor cooper-floor))
                (not (= fletcher-floor baker-floor))
                (not (= fletcher-floor 1))
                (not (= fletcher-floor 5))
                (not (= (abs (- fletcher-floor cooper-floor)) 1))))
          ((= who miller)
           (and (not (= miller-floor fletcher-floor))
                (not (= miller-floor cooper-floor))
                (not (= miller-floor baker-floor))
                (> miller-floor cooper-floor)))
          ((= who smith)
           (and (not (= smith-floor miller-floor))
                (not (= smith-floor fletcher-floor))
                (not (= smith-floor cooper-floor))
                (not (= smith-floor baker-floor))
                (not (= (abs (- smith-floor fletcher-floor)) 1))))
          (else (error "invalid dweller" who)))))

(define (try-dwell)
  (define (place who floor result)
    (if (> who smith)
        (display result)
        (let ((next (cons floor result)))
          (if (check next who)
              (place (+ who 1) 1 next)
              (if (< floor 5)
                  (place who (+ floor 1) result)))))))
  (place 0 1 '()))
(try-dwell)
; (1 5 4 2 3)
```

xdaavidliu

Simple and very efficient solution that treats (list fletcher smith cooper miller baker) as a big-endian base-5 number (with possible digits 1-5 instead of 0-4), which we then iterate through in order.

```

(define (nearby? j k)
  (>= 1 (abs (- j k)))))

(define (ordinary-multiple-dwelling)
  (define (display-solution f s c m b)
    (display
      (list (list 'baker b) (list 'cooper c)
            (list 'fletcher f) (list 'miller m)
            (list 'smith s)))
    (newline))
  (define (iter-f f)
    (cond ((= f 1) (iter-f 2))
          ((= f 5) 'done)
          (else (iter-s f 1))))
  (define (iter-s f s)
    (cond ((> s 5) (iter-f (1+ f)))
          ((nearby? f s) (iter-s f (1+ s))) ; see additional note
          (else (iter-c f s 1))))
  (define (iter-c f s c)
    (cond ((> c 5) (iter-s f (1+ s)))
          ((or (nearby? f c) (= c 1) (= c s))
           (iter-c f s (1+ c)))
          (else (iter-m f s c 1))))
  (define (iter-m f s c m)
    (cond ((> m 5) (iter-c f s (1+ c)))
          ((or (<= m c) (= m s) (= m f))
           (iter-m f s c (1+ m)))
          (else (iter-b f s c m 1))))
  (define (iter-b f s c m b)
    (cond ((> b 5) (iter-m f s c (1+ m)))
          (else (if (not (or (= b 5) (= b m) (= b c) (= b s) (= b f)))
                    (display-solution f s c m b))
                  (iter-b f s c m (1+ b))))))
  (iter-f 1))

(ordinary-multiple-dwelling)
;; ((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))
;Value: done

```

Additional note: if we change the predicate (nearby? f s) to (= f s) and re-run this procedure, we obtain the extra solutions from exercise 4.38:

```

(ordinary-multiple-dwelling)
;; ((baker 3) (cooper 4) (fletcher 2) (miller 5) (smith 1))
;; ((baker 1) (cooper 4) (fletcher 2) (miller 5) (smith 3))
;; ((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))
;; ((baker 1) (cooper 2) (fletcher 4) (miller 5) (smith 3))
;; ((baker 1) (cooper 2) (fletcher 4) (miller 3) (smith 5))
;; ;Value: done

```

meteorgan

```

(define (flatmap proc li)
  (if (null? li)
      '()
      (let ((result (proc (car li)))
            (rest (flatmap proc (cdr li))))
        (if (pair? result)
            (append result rest)
            (cons result rest)))))

(define (permutations lists)
  (if (null? lists)
      '()
      (flatmap (lambda (x)
                 (map (lambda (y) (cons x y))
                      (permutations (cdr lists))))
              (car lists)))))

(define (restrictions l)
  (apply
    (lambda (baker cooper fletcher miller smith)
      (and (> miller cooper)
           (not (= (abs (- smith fletcher)) 1))
           (not (= (abs (- fletcher cooper)) 1))
           (distinct? (list baker cooper fletcher miller smith))))
    l))

(define (mutiple-dwelling)
  (let ((baker '(1 2 3 4))
        (cooper '(2 3 4 5)))

```

```
(fletcher '(2 3 4))
(miller '(3 4 5))
(smith '(1 2 3 4 5)))
(filter restrictions (permutations (list baker cooper fletcher miller smith))))
```

Felix021

another solution, closer to the original amb program.

```
(define (multiple-dwelling)
  (define (flat-map proc lst)
    (if (null? lst)
        '()
        (let ((first (proc (car lst))))
          ((if (pair? first) append cons)
           first
           (flat-map proc (cdr lst))))))

  (define (permutations lst)
    (if (null? lst)
        (list '())
        (flat-map
         (lambda (first)
           (map
            (lambda (rest)
              (cons first rest))
            (permutations (filter (lambda (x) (not (= x first))) lst))))
           lst)))
    (for-each
     (lambda (try)
       (apply
        (lambda (baker cooper fletcher miller smith)
          (if (and (!= baker 5)
                    (!= cooper 1)
                    (!= fletcher 1)
                    (!= fletcher 5)
                    (> miller cooper)
                    (!= (abs (- smith fletcher)) 1)
                    (!= (abs (- fletcher cooper)) 1))
            (display (list baker cooper fletcher miller smith))))
          try))
       (permutations '(1 2 3 4 5)))))

  (multiple-dwelling))
```

Shaw

An ugly solution.

```
(define (flatmap f lst)
  (if (null? lst)
      null
      (let ((result (f (car lst)))
            (rest (flatmap f (cdr lst))))
        (if (or (pair? result) (null? result))
            (append result rest)
            (cons result rest)))))

(define (distinct? l)
  (define (member? a lst)
    (cond ((null? lst) #f)
          ((eq? (car lst) a) #t)
          (else (member? a (cdr lst)))))

  (cond
   ((null? l) #t)
   ((member? (car l) (cdr l)) #f)
   (else (distinct? (cdr l)))))

(define (solve)
  (flatmap
   (lambda (cooper)
     (flatmap
      (lambda (baker)
        (flatmap
         (lambda (fletcher)
           (if (= (abs (- cooper fletcher)) 1)
               null
               (flatmap
                (lambda (miller)
```

```

(if (not (> miller cooper))
    null
    (flatmap
        (lambda (smith)
            (if (and (not (= (abs (- smith fletcher)) 1))
                      (distinct? (list
                                    cooper baker fletcher miller smith)))
                    (list (list 'cooper cooper)
                          (list 'baker baker)
                          (list 'fletcher fletcher)
                          (list 'miller miller)
                          (list 'smith smith)))
                null)
            ' (1 2 3 4 5))))
        ' (3 4 5)))
    ' (2 3 4)))
  ' (1 2 3 4))
  ' (2 3 4 5)))
;; ((cooper 2) (baker 3) (fletcher 4) (miller 5) (smith 1))

```

```

(define (solve)
  (let ((result '()))
    (map
      (lambda (cooper)
        (map
          (lambda (baker)
            (map
              (lambda (fletcher)
                (if (= (abs (- cooper fletcher)) 1)
                    null
                    (map
                      (lambda (miller)
                        (if (not (> miller cooper))
                            null
                            (map
                              (lambda (smith)
                                (if (and (not (= (abs (- smith fletcher)) 1))
                                      (distinct? (list
                                                    cooper baker fletcher miller smith)))
                                    (set! result
                                          (cons
                                            (list (list 'cooper cooper)
                                                  (list 'baker baker)
                                                  (list 'fletcher fletcher)
                                                  (list 'miller miller)
                                                  (list 'smith smith))
                                            result)))
                                    null))
                                ' (1 2 3 4 5)))))
                    ' (3 4 5))))
                  ' (2 3 4)))
                ' (1 2 3 4)))
              ' (2 3 4 5))
            (display result)))
  ;(((cooper 2) (baker 3) (fletcher 4) (miller 5) (smith 1)))

```

```

        (list 'miller m)
        (list 'fletcher f)
        (list 'smith s))
      result))
  (iter-s (+ s 1))))
  (cond ((or (= f b) (= f c) (= f m) (= (abs (- f c)) 1))
    (iter-f (+ f 1)))
    ((> f 4)
     (iter-m (+ m 1)))
    (else (iter-s 1))))
  (cond ((or (= m b) (<= m c))
    (iter-m (+ m 1)))
    ((> m 5)
     (iter-c (+ c 1)))
    (else (iter-f 2))))
  (cond ((= c b)
    (iter-c (+ c 1)))
    ((> c 5)
     (iter-b (+ b 1)))
    (else (iter-m 3))))
  (cond ((> b 4)
    result)
    (else (iter-c 2))))
  (iter-b 1)))
;;(((baker 3) (cooper 2) (miller 5) (fletcher 4) (smith 1))

```

stepvhen

Instead of generating permutations, we can consider the separate dwellings as a 5 digit base-5 number, and with each pass we increment that number.

```

(define (multiple-dwellings)
  (define (house-iter b c m f s)
    (cond ((> b 4) ; Baker can't live on 5th floor.
           '(no answer available))
          ((> c 5)
           (house-iter (+ b 1) 2 3 2 1))
          ((> m 5)
           (house-iter b (+ c 1) (+ c 2) 2 1)) ; miller is above cooper
          ((> f 4) ; fletcher can't live on 5th floor
           (house-iter b c (+ m 1) 2 1))
          ((> s 5)
           (house-iter b c m (+ f 1) 1))
          ((and (not (= (abs (- s f)) 1))
                 (not (= (abs (- c f)) 1))
                 (distinct? (list b c m f s)))
            (list (list 'baker b) (list 'cooper c)
                  (list 'fletcher f) (list 'miller m)
                  (list 'smith s)))
          (else
           (house-iter b c m f (+ s 1)))))
    (house-iter 1 2 3 2 1)) ; initial values take some restrictions into account

```

timothy235

Racket has great list comprehensions.

```

(define (solution? baker cooper fletcher miller smith)
  (and ; (distinct? (list baker cooper fletcher miller smith))
       (not (= baker 5))
       (not (= cooper 1))
       (not (= fletcher 5))
       (not (= fletcher 1))
       (> miller cooper)
       (not (= (abs (- smith fletcher)) 1))
       (not (= (abs (- fletcher cooper)) 1)))))

(define (show-solutions)
  (for/list ([tenants (in-permutations (range 1 6))])
    #:when (apply solution? tenants)
    (map list
         '(baker cooper fletcher miller smith)
         tenants)))

(show-solutions)
;; (((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1)))

```

revc

A solution that is easy to read or understand.

```

;; aliases for Chez Scheme
(define false #f)
(define true #t)

(define (multiple-dwelling)
  (define ans '())
  (define names (list 'baker 'cooper 'fletcher 'miller 'smith))

  ;; selectors
  (define (baker assignment) (list-ref assignment 0))
  (define (cooper assignment) (list-ref assignment 1))
  (define (fletcher assignment) (list-ref assignment 2))
  (define (miller assignment) (list-ref assignment 3))
  (define (smith assignment) (list-ref assignment 4))

  (define (distinct? items)
    (cond ((null? items) true)
          ((null? (cdr items)) true)
          ((member (car items) (cdr items)) false)
          (else (distinct? (cdr items)))))

  ;; is an satiable assignment?
  (define (satisfiable? assignment)
    (and (distinct?
          (list (baker assignment) (cooper assignment) (fletcher assignment) (miller
assignment) (smith assignment)))
    (not (= (baker assignment) 5))
    (not (= (cooper assignment) 1))
    (not (= (fletcher assignment) 5))
    (not (= (fletcher assignment) 1))
    (> (miller assignment) (cooper assignment))
    ;; (not (= (abs (- (smith assignment) (fletcher assignment))) 1))
    (not (= (abs (- (fletcher assignment) (cooper assignment))) 1)))))

  ;; try with experimental assignment at the specified stage.
  (define (try r-assignment stage)
    (cond [(< stage 5)
           (let loop ([floor 1])
             (if (< floor 6)
                 (begin (try (cons floor r-assignment) (+ stage 1))
                        (loop (+ floor 1))))
               [ (= stage 5) (if (satisfiable? (reverse r-assignment))
                                (set! ans (cons (reverse r-assignment) ans))))]))]
         (try '() 0)

  ;; combine names with floors
  ;; ``reverse`` is optional
  (reverse (map (lambda (assignment) (map list names assignment)) ans)))

```

>>>Thomas simple solution. Just filter from permutations

```

(define (multiple-dwelling)
  ;;helper procedures
  (define (remove x s)
    (filter (lambda(y) (not (eq? x y))) s))
  (define (permutations list)
    (if (null? list) '()
        (flatmap (lambda (x) (map (lambda (y) (cons x y)) (permutations (remove x list))))
list)))
  (define (accumulate proc int list)
    (if (null? list) int
        (proc (car list) (accumulate proc int (cdr list)))))
  (define (flatmap proc list)
    (accumulate append '() (map proc list)))
  (define (list-position obj list)
    (define (search rem-list n)
      (if (null? rem-list) (length list)
          (if (eq? (car rem-list) obj) n
              (search (cdr rem-list) (+ n 1)))))
    (search list 1))
  (define (higher? A B list)
    (> (list-position A list) (list-position B list)))
  (define (adjacent? A B list)
    (= 1 (abs (- (list-position A list) (list-position B list)))))

  (define (filter predicate list)
    (if (null? list) '()
        (if (predicate (car list))
            (cons (car list) (filter predicate (cdr list)))
            (filter predicate (cdr list)))))

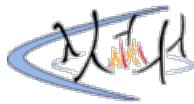
  ;;actual procedure
  (filter (lambda (list)
            (let ((first (list-ref list 0))
                  (sec (list-ref list 1))
                  (third (list-ref list 2)))

```

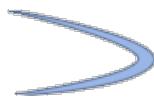
```
(fourth (list-ref list 3))
(fifth (list-ref list 4)))
(and (not (eq? fifth 'B))
(not (eq? first 'C))
(not (or (eq? first 'F) (eq? fifth 'F)))
(higher? 'M 'C list)
(not (adjacent? 'S 'F list))
(not (adjacent? 'F 'C list))))) (permutations (list 'B 'C 'F 'M
'S))))
```

---

Last modified : 2020-05-04 06:56:21  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-4.42



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.41) | Index | Next exercise (4.43) >>

woofy

Solution without permutation.

```
(define stmts
  '(((kitty 2) (betty 3))
    ((ethel 1) (joan 2))
    ((joan 3) (ethel 5))
    ((kitty 2) (betty 1))
    ((mary 4) (betty 1)))))

(define (no-conflict kv set)
  (define (iter next)
    (or (null? next)
        (let ((k (caar next)))
          (v (cadar next)))
        (if (= (car kv) k)
            (and (= v (cadr kv)) (iter (cdr next)))
            (iter (cdr next))))))
  (iter set))

(define (choose girl-says)
  (define (iter rest-girl-says selected)
    (if (null? rest-girl-says)
        (display selected)
        (let ((s1 (caar rest-girl-says))
              (s2 (cadar rest-girl-says)))
          (let ((which (amb s1 s2)))
            (require (no-conflict which selected))
            (iter (cdr rest-girl-says) (cons which selected))))))
  (iter girl-says '())))

(choose stmts)
```

meteorgan

```
(define (lier)
  (let ((a (amb 1 2 3 4 5))
        (b (amb 1 2 3 4 5))
        (c (amb 1 2 3 4 5))
        (d (amb 1 2 3 4 5))
        (e (amb 1 2 3 4 5)))
    (require (or (and (= d 2) (not (= a 3))) (and (not (= d 2)) (= a 3))))
            (require (or (and (= b 1) (not (= c 2))) (and (not (= b 1)) (= c 2))))
                    (require (or (and (= c 3) (not (= b 5))) (and (not (= c 3)) (= b 5))))
                            (require (or (and (= d 2) (not (= e 4))) (and (not (= d 2)) (= e 4)))
                                (require (or (and (= e 4) (not (= a 1))) (and (not (= e 4)) (= a 1)))
                                    (require (distinct? (list a b c d e)))
                                    (list a b c d e)))
                    result is: (3 5 2 1 4))
```

3pmtea

With a utility procedure "xor" defined like below, the solution to this exercise would look more elegant. I don't use "and" and "or" in its implementation, since a lot of work has to be done in order to make the amb-interpreter support them.

```
(define (xor p q) (if p (not q) q))
```

thanhnghuyen2187

and and or can be added to the evaluator like this:

```
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)))
```

```

(list 'cons cons)
(list 'null? null?)
(list 'list list)
(list 'memq memq)
(list 'member member)
(list 'not not)
(list '+ +)
(list '- -)
(list '* *)
(list '= =)
(list '> >)
(list '>= >=)
(list 'abs abs)
(list 'remainder remainder)
(list 'integer? integer?)
(list 'sqrt sqrt)
(list 'eq? eq?)

;; more primitives
(list 'and (lambda (clause-1 clause-2) (and clause-1 clause-2)))
(list 'or (lambda (clause-1 clause-2) (or clause-1 clause-2)))
())

```

Shaw

```

(define (flatmap f lst)
  (if (null? lst)
      null
      (let ((result (f (car lst))))
        (rest (flatmap f (cdr lst))))
      (if (or (pair? result) (null? result))
          (append result rest)
          (cons result rest)))))

(define (permuta lst)
  (if (null? lst)
      '()
      (flatmap
        (lambda (x)
          (map
            (lambda (y)
              (cons x y))
            (permuta (remove x lst))))
        lst)))

(define (remove a lst)
  (cond
    ((null? lst) null)
    ((eq? a (car lst)) (cdr lst))
    (else (cons (car lst)
                 (remove a (cdr lst))))))

(define first car)
(define second cadr)
(define third caddr)
(define fourth cadddr)
(define (fifth x)
  (car (cddddr x)))

(define (xor a b)
  (and (or a b)
       (not (and a b)))))

(define betty-restrictions
  (lambda (lst)
    (xor (eq? (second lst) 'kitty)
         (eq? (third lst) 'betty)))))

(define ethel-restrictions
  (lambda (lst)
    (xor (eq? (first lst) 'ethel)
         (eq? (second lst) 'john)))))

(define john-restrictions
  (lambda (lst)
    (xor (eq? (third lst) 'john)
         (eq? (fifth lst) 'ethel)))))


```

```

(define kitty-restrictions
  (lambda (lst)
    (xor (eq? (second lst) 'kitty)
         (eq? (fourth lst) 'mary)))))

(define mary-restrictions
  (lambda (lst)
    (xor (eq? (fourth lst) 'mary)
         (eq? (first lst) 'betty)))))

(define restrictions-lists
  (list betty-restrictions
        ethel-restrictions
        john-restrictions
        kitty-restrictions
        mary-restrictions))

(define name-lists
  (permuta '(betty ethel john kitty mary)))

(define (pass-all? tests ele)
  (if (null? tests)
      #t
      (and ((car tests) ele)
           (pass-all? (cdr tests) ele)))))

(define (filter f lst)
  (cond
    ((null? lst) null)
    ((f (car lst)) (cons (car lst) (filter f (cdr lst))))
    (else
      (filter f (cdr lst)))))

(filter (lambda (x) (pass-all? restrictions-lists x))
       name-lists)

;;((kitty john betty mary ethel))

```

codybartfast

```

(define (require-one p q)
  (require (if p (not q) q)))

(define (liars-puzzle)
  (let ((betty (amb 1 2 3 4 5))
        (ethel (amb 1 2 3 4 5))
        (joan (amb 1 2 3 4 5))
        (kitty (amb 1 2 3 4 5))
        (mary (amb 1 2 3 4 5)))
    (require-one (= kitty 2) (= betty 3))
    (require-one (= ethel 1) (= joan 2))
    (require-one (= joan 3) (= ethel 5))
    (require-one (= kitty 2) (= mary 4))
    (require-one (= mary 4) (= betty 1))
    (require (distinct? (list betty ethel joan kitty mary)))
    (list (list 'betty betty)
          (list 'ethel ethel)
          (list 'joan joan)
          (list 'kitty kitty)
          (list 'mary mary))))

```

>>> Thomas (04-2020) since filter checks every permutation we just need to make sure that each statement contains one true and one false clause. (we don't need to worry that the clauses are coherent, since we're checking a given list)

```

(define (liar)
  ;;helper procedures
  (define (filter predicate list)
    (if (null? list) '()
        (if (predicate (car list))
            (cons (car list) (filter predicate (cdr list)))
            (filter predicate (cdr list)))))

  (define (accumulate proc int list)
    (if (null? list) int
        (proc (car list) (accumulate proc int (cdr list)))))

  (define (flatmap proc list)
    (accumulate append '() (map proc list)))

  (define (remove x s)
    (filter (lambda (y) (not (eq? x y))) s)))

```

```

(define (permutations list)
  (if (null? list) '()
      (flatmap (lambda (x) (map (lambda (y) (cons x y)) (permutations (remove x list))))
list)))
(define (xor a b)
(and (or a b) (not (and a b))))
;actual procedure
(filter (lambda (list)
  (let ((first (list-ref list 0))
        (sec (list-ref list 1))
        (third (list-ref list 2))
        (fourth (list-ref list 3))
        (fifth (list-ref list 4)))
    (and (xor (eq? sec 'K) (eq? third 'B))
         (xor (eq? first 'E) (eq? sec 'J))
         (xor (eq? third 'J) (eq? fifth 'E))
         (xor (eq? sec 'K) (eq? fourth 'M))
         (xor (eq? fourth 'M) (eq? first 'B))))))
  (permutations (list 'B 'E 'J 'K 'M))))

```

tiendo1011

I think instead of starting with each person can take up any place from (1 2 3 4 5). We can just listen to there statements about each other, choose one from each and see if it produces a sensible result:

```

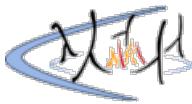
(define (liars)
(let ((betty (amb 3 1))
(ethel (amb 1 5))
(joan (amb 3 2))
(kitty (amb 2))
(mary (amb 4)))
(require
  (distinct? (list betty ethel joan kitty mary)))
(list (list 'betty betty)
      (list 'ethel ethel)
      (list 'joan joan)
      (list 'kitty kitty)
      (list 'mary mary)))
(liars)

```

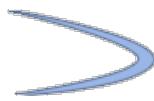
The result is: ((betty 1) (ethel 5) (joan 3) (kitty 2) (mary 4))

master

Note that this answer cannot possibly be correct. If Betty is first that means Mary and Kitty were lying about Mary being fourth, which means that Kitty is second. But if Kitty is second then Joan can't also be second which means that Ethel was lying about Joan being second which means that Ethel was actually first. So assuming that Betty was first led us to a contradiction. It's actually easier to demonstrate that there's a contradiction by noting that Kitty being second and Mary being fourth cannot simultaneously be true because Kitty made both of those claims so one of them must be false. The correct answer is ((betty 3) (ethel 5) (joan 2) (kitty 1) (mary 4)).



# sicp-ex-4.43



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.42) | Index | Next exercise (4.44) >>

woofy

```
; a. Lorna's father is Colonel Downing

; b.
(define yacht-names
  ' ((Moore Lorna)
    (Downing Melissa)
    (Barnacle Gabrielle)
    (Hall Rosalind)
    (Parker Mary)))

(define daddy-options
  ' ((Lorna (Downing Barnacle Hall Parker))
    (Melissa (Barnacle))
    (Gabrielle (Moore Downing Hall Parker)))
    (Rosalind (Moore Downing Parker))
    (Mary (Moore Downing Barnacle Hall)))))

(define (find-cdr kv-list k)
  (define (iter rest)
    (cond ((null? rest) false)
          ((eq? k (caar rest)) (cadar rest))
          (else (iter (cdr rest)))))
  (iter kv-list))

(define (find-car kv-list k)
  (define (iter rest)
    (cond ((null? rest) false)
          ((eq? k (cadar rest)) (caar rest))
          (else (iter (cdr rest)))))
  (iter kv-list))

(define (daddy girl set) (find-cdr set girl))
(define (daughter daddy set) (find-car set daddy))
(define (yacht daddy set) (find-cdr set daddy))

(define (no-share-daddy daddy set)
  (define (iter rest)
    (cond ((null? rest) true)
          ((eq? (cadar rest) daddy) false)
          (else (iter (cdr rest)))))
  (iter set))

(define (nice-daughters)
  (define (iter options result)
    (if (null? options)
        (begin
          (require (eq? (yacht (daddy 'Gabrielle)) (daughter 'Parker)))
          result)
        (let ((girl (caar options))
              (daddies (cadar options)))
          (let ((daddy (a-element-of daddies)))
            (require (no-share-daddy daddy result))
            (iter (cdr options) (cons (list girl daddy) result)))))))
  (iter daddy-options '()))
```

xdaividliu

Here's a solution that doesn't explicitly leave out solutions, instead having amb and require eliminate them naturally.

```
(define (yacht)
  (define gab 'gabrielle)
  (define lor 'lorna)
  (define ros 'rosalind)
  (define mel 'melissa)
  (define mar 'mary-ann)
  (let ((barnacle (amb gab lor ros mel mar)))
```

```

(require (eq? barnacle mel))
(let ((moore (amb gab lor ros mel mar)))
  (require (eq? moore mar))
  (let ((hall (amb gab lor ros mel mar)))
    (require (not (memq hall (list barnacle moore ros))))
    (let ((downing (amb gab lor ros mel mar)))
      (require (not (memq downing (list barnacle moore hall mel))))
      (let ((parker (amb gab lor ros mel mar)))
        (require (not (memq parker
          (list barnacle moore hall downing mar))))
        (let ((yacht-names
          (list (list barnacle gab)
            (list moore lor)
            (list hall ros)
            (list downing mel)
            (list parker mar))))
          (require (eq? parker (cdr (assq gab yacht-names)))))
          (list (list 'barnacle barnacle)
            (list 'moore moore)
            (list 'hall hall)
            (list 'downing downing)
            (list 'parker parker)))))))))))

```

meteorgan

```

(define (father-daughter)
  (let ((Moore 'Mary)
    (Barnacle 'Melissa)
    (Hall (amb 'Gabrielle 'Lorna))
    (Downing (amb 'Gabrielle 'Lorna 'Rosalind))
    (Parker (amb 'Lorna 'Rosalind)))
  (require (cond ((eq? Hall 'Gabrielle) (eq? 'Rosalind Parker))
    ((eq? Downing 'Gabrielle) (eq? 'Melissa Parker))
    (else false)))
  (require (distinct? (list Hall Downing Parker)))
  (list (list 'Barnacle Barnacle)
    (list 'Moore Moore)
    (list 'Hall Hall)
    (list 'Downing Downing)
    (list 'Parker Parker)))))

run (father-daughter), get ((Barnacle Melissa) (Moore Mary) (Hall Gabrielle) (Downing Lorna) (Parker Rosalind)), so Lorna's father is Colonel Downing.
If we don't know Mary Ann's family name is Moore, we get:
;; Starting a new problem
;; Amb-Eval output:
((Barnacle Melissa) (Moore Gabrielle) (Hall Mary) (Downing Rosalind) (Parker Lorna))

;; Amb-Eval input:
try-again

;; Amb-Eval output:
((Barnacle Melissa) (Moore Mary) (Hall Gabrielle) (Downing Lorna) (Parker Rosalind))

```

davl

To express `Gabrielle's father owns the yacht that is named after Dr. Parker's daughter', I introduce 2 procedures `name-of-his-yacht' and `her-father'

```

(define (game-of-yacht)
  (define (all-fathers) (amb 'mr-moore 'colonel-downing 'mr-hall 'sir-barnacle-hood 'dr-parker))

  (define (all-fathers-except except-father)
    (let ((fathers (all-fathers)))
      (require (not (eq? fathers except-father)))
      fathers))

  (define (name-of-his-yacht father-name)
    (cond ((eq? father-name 'mr-moore) 'lorna)
      ((eq? father-name 'colonel-downing) 'melissa)
      ((eq? father-name 'mr-hall) 'rosalind)
      ((eq? father-name 'sir-barnacle-hood) 'gabrielle)
      ((eq? father-name 'dr-parker) 'marry))) ; a little jumpy here

  (define lorna-father (all-fathers-except 'mr-moore))
  (define melissa-father 'sir-barnacle-hood)
  (define rosalind-father (all-fathers-except 'mr-hall))
  (define gabrielle-father (all-fathers-except 'sir-barnacle-hood))
  (define marry-father 'mr-moore))

```

```

(define (her-father she)
  (cond ((eq? she 'lorna) lorna-father)
        ((eq? she 'melissa) melissa-father)
        ((eq? she 'rosalind) rosalind-father)
        ((eq? she 'gabrielle) gabrielle-father)
        ((eq? she 'mary) mary-father)))

(require (distinct? (list lorna-father melissa-father rosalind-father gabrielle-father
                           mary-father)))
(require (eq? (her-father (name-of-his-yacht gabrielle-father)) 'dr-parker))
(list lorna-father melissa-father rosalind-father gabrielle-father mary-father))

```

Output:

```

(game-of-yacht)
;; Starting a new problem
;; Amb-Eval value:
(colonel-downing sir-barnacle-hood dr-parker mr-hall mr-moore)

try-again
;; There are no more values of
(game-of-yacht)

```

revc

self-documenting code.

```

(define (list-ref lst n)
  (if (= n 0)
      (car lst)
      (list-ref (cdr lst) (- n 1)))

;; single let but inefficient
(define (Yacht)
  (define moore 1)
  (define downing 2)
  (define hall 3)
  (define barnacle 4)
  (define parker 5)

  (define lorna 1)
  (define mellissa 2)
  (define rosalind 3)
  (define gabrille 4)
  (define mary 5)

  (define (father-name father)
    (define names (list 'moore 'downing 'hall 'barnacle 'parker))
    (list-ref names (- father 1)))

  (define (owned-yacht father)
    father)

  (define (father-of daughter fathers)
    (list-ref fathers (- daughter 1)))

  (let ((F-lorna (amb moore downing hall barnacle parker))
        (F-mellissa (amb moore downing hall barnacle parker))
        (F-rosalind (amb moore downing hall barnacle parker))
        (F-gabrille (amb moore downing hall barnacle parker))
        (F-mary (amb moore downing hall barnacle parker)))
    (let ((fathers (list F-lorna F-mellissa F-rosalind F-gabrille F-mary)))
      (require (not (= F-lorna moore)))
      (require (not (= F-rosalind hall)))
      (require (not (= F-gabrille barnacle)))
      (require (= F-mellissa barnacle))
      (require (= F-mary moore))
      (require (= (father-of (owned-yacht F-gabrille) fathers) parker))
      (require
       (distinct? fathers))
      (list (list 'lorna (father-name F-lorna))
            (list 'mellissa (father-name F-mellissa))
            (list 'rosalind (father-name F-rosalind))
            (list 'gabrille (father-name F-gabrille))
            (list 'mary (father-name F-mary)))))

;; nested let but but efficient
(define (Yacht)
  (define moore 1)
  (define downing 2)
  (define hall 3)
  (define barnacle 4)
  (define parker 5)

```

```

(define lorna 1)
(define mellissa 2)
(define rosalind 3)
(define gabrille 4)
(define mary 5)

(define (father-name father)
  (define names (list 'moore 'downing 'hall 'barnacle 'parker))
  (list-ref names (- father 1)))

(define (owned-yacht father)
  father)

(define (father-of daughter fathers)
  (list-ref fathers (- daughter 1)))

(let ([F-mary (amb moore downing hall barnacle parker)])
  (require (= F-mary moore))
  (let ([F-mellissa (amb moore downing hall barnacle parker)])
    (require (= F-mellissa barnacle))
    (let ([F-lorna (amb moore downing hall barnacle parker)])
      (require (not (= F-lorna moore)))
      (let ([F-rosalind (amb moore downing hall barnacle parker)])
        (require (not (= F-rosalind hall)))
        (let ([F-gabrille (amb moore downing hall barnacle parker)])
          (require (not (= F-gabrille barnacle)))
          (let ((fathers (list F-lorna F-mellissa F-rosalind F-gabrille F-mary)))
            (require (= (father-of (owned-yacht F-gabrille) fathers) parker))
            (require (distinct? fathers))
            (list (list 'lorna (father-name F-lorna))
                  (list 'mellissa (father-name F-mellissa))
                  (list 'rosalind (father-name F-rosalind))
                  (list 'gabrille (father-name F-gabrille))
                  (list 'mary (father-name F-mary)))))))))))

```

Thomas (04-2020)

Here's a solutions in normal scheme (had to change the structure to make it readable)

```

(define (puzzle)
;;internal helper definition
(define (filter predicate list)
  (if (null? list) '()
    (if (predicate (car list))
        (cons (car list) (filter predicate (cdr list)))
        (filter predicate (cdr list)))))

(define (accumulate proc int list)
  (if (null? list) int
    (proc (car list) (accumulate proc int (cdr list)))))

(define (flatmap proc list)
  (accumulate append '() (map proc list)))

(let ((all-triples ;;form: dad yacht daughter - gives a list whos sublists are all
       ;triples with one of the dads (5 sublist- in total 36 triples)
       (map (lambda (dad)
              (filter (lambda (x) (if (eq? x 'yacht=daughter) false ;in order to get
                     a shorter list, we filter whats possible directly
                     (and (if (eq? dad 'B) (if (eq? (cadr x) 'G) true false) true) ;;
                     Barnacles yacht is gabriele
                     (if (eq? dad 'M) (if (eq? (cadr x) 'L) true false) true)
                     ;Moores is Lorna
                     (if (eq? dad 'H) (if (eq? (cadr x) 'R) true false) true)
                     ;Halls is Rosalinda
                     (if (eq? dad 'D) (if (eq? (cadr x) 'M) true false)) true))) ;; Downings is Melissa
              (flatmap (lambda (yacht)
                         (map (lambda (daughter)
                                (if (eq? yacht daughter) 'yacht=daughter
                                    (list dad yacht daughter)))
                         (list 'G 'L 'R 'A 'M))) ;;daughter names are
                         (list 'G 'L 'R 'A 'M))) ;;also yacht names
                         (list 'B 'M 'H 'P 'D)))) ;abbreviations for dad names
       (let ((all-combinations ;;gives a list with all combinations of triples (with the
             ;incorperated restrictions 44 in total)
             (let ((all all-triples))
               (let ((triples-1dad (car all))
                     (triples-2dad (cadr all))
                     (triples-3dad (caddr all))
                     (triples-4dad (cadddr all))
                     (triples-5dad (car (cddddr all))))
                 (flatmap (lambda (1dad)
                            (flatmap (lambda (2dad)
                                       (flatmap (lambda (3dad)
                                                 (flatmap (lambda (4dad)
                                                       (list (list 1dad 2dad 3dad 4dad))))))))))))))) ;; we want to filter out all combinations where the dady have same yachts or daughters

```

```

(filter (lambda (x) (let ((1dady (cadr (car x)))
  (1dadd (caddr (car x)))) ;we therefor abbreviate (dadd dad-daughter) (dady dad-
yacht)
  (2dady (cadr (cadr x)))
  (2dadd (caddr (cadr x)))
  (3dady (cadr (caddr x)))
  (3dadd (caddr (caddr x)))
  (4dady (cadr (cadddr x)))
  (4dadd (caddr (cadddr x)))
  (5dady (cadar (cddddr x)))
  (5dadd (cadar (cddddr x))))
  (if (or (eq? 1dady 2dady) (eq? 1dady 3dady) (eq? 1dady 4dady) (eq?
1dady 5dady)
    (eq? 2dady 3dady) (eq? 2dady 4dady) (eq? 2dady 5dady)
    (eq? 3dady 4dady) (eq? 3dady 5dady) (eq? 4dady 5dady)
    (eq? 1dadd 2dadd) (eq? 1dadd 3dadd) (eq? 1dadd 4dadd) (eq?
1dadd 5dadd)
      (eq? 2dadd 3dadd) (eq? 2dadd 4dadd) (eq? 2dadd 5dadd)
      (eq? 3dadd 4dadd) (eq? 3dadd 5dadd) (eq? 4dadd 5dadd)) false
true)))
  (map (lambda (5dad) (list 1dad 2dad 3dad 4dad 5dad)) tripples-5dad)))
  tripples-4dad))
  tripples-3dad))
  tripples-2dad))
  tripples-1dad)))))

(filter (lambda (combination)
  (let ((B-tripple (car combination))
        (M-tripple (cadr combination))
        (H-tripple (caddr combination))
        (P-tripple (cadddr combination))
        (D-tripple (car (cddddr combination)))))

  (if (and (eq? (cadr D-tripple) (caddr B-tripple)) ; yacht from Downing is Barnacles
daughter
    (eq? (caddr M-tripple) 'A);; Ann is Moores daughter
    (cond ((eq? (caddr B-tripple) 'G) (eq? (cadr B-tripple) (caddr P-tripple))) ;the one
who's Gabrielle's father owns the yacht that parkers daughter
      ((eq? (caddr M-tripple) 'G) (eq? (cadr M-tripple) (caddr P-tripple)))
      ((eq? (caddr H-tripple) 'G) (eq? (cadr H-tripple) (caddr P-tripple)))
      ((eq? (caddr P-tripple) 'G) (eq? (cadr P-tripple) (caddr P-tripple)))
      ((eq? (caddr D-tripple) 'G) (eq? (cadr D-tripple) (caddr P-tripple)))
      true false)))
    true false))) all-combinations)))))

;;result: (((B G M) (M L A) (H R G) (P A R) (D M L)))
;;if we erase the restriction that Ann is Moores daughter, we get two possibilities
;; result: (((B G M) (M L G) (H R A) (P A L) (D M R)) ((B G M) (M L A) (H R G) (P A R) (D
M L)))

```

SteeleDynamic

To simplify the implementation, define an internal procedure that maps the surname of the yacht owner to the surname of the yacht name. This procedure needs to be redefined each time any of the ambiguous variables' values change. This redefinition occurs automatically if the internal procedure definition occurs in the body of the 'let' expression. Below is the nondeterministic evaluator input and output:

```
; nondeterministic evaluator implementation...

1 |=> (define the-global-environment (setup-environment))
;Value: the-global-environment

1 |=> (driver-loop)

;; Amb-Eval input:
(define (require p)
  (if (not p) (amb)))
;; Starting a new problem
;; Amb-Eval value:
ok

;; Amb-Eval input:
(define (distinct? items)
  (cond ((null? items) true)
        ((null? (cdr items)) true)
        ((member (car items) (cdr items)) false)
        (else (distinct? (cdr items)))))
;; Starting a new problem
;; Amb-Eval value:
ok

;; Amb-Eval input:
(define (daughters1)
  (let ((gabrielle (amb 'downing 'hall 'parker))
        (lorna (amb 'downing 'hall 'parker))
        (mary 'moore))
```

```

(melissa 'hood)
(rosalind (amb 'downing 'hall 'parker)))
(define (yacht-of father)
  (cond ((eq? father 'downing) melissa)
        ((eq? father 'hall) rosalind)
        ((eq? father 'hood) gabrielle)
        ((eq? father 'moore) lorna)
        (else mary)))
(require (eq? 'parker (yacht-of gabrielle)))
(require (distinct? (list gabrielle lorna mary melissa rosalind)))
(list (list 'gabrielle gabrielle)
      (list 'lorna lorna)
      (list 'mary mary)
      (list 'melissa melissa)
      (list 'rosalind rosalind)))
;;; Starting a new problem
;;; Amb-Eval value:
ok

;;; Amb-Eval input:
(daughters1)
;;; Starting a new problem
;;; Amb-Eval value:
((gabrielle hall) (lorna downing) (mary moore) (melissa hood) (rosalind parker))

;;; Amb-Eval input:
try-again
;;; There are no more values of
(daughters1)

;;; Amb-Eval input:
(define (daughters2)
  (let ((gabrielle (amb 'downing 'hall 'moore 'parker))
        (lorna (amb 'downing 'hall 'moore 'parker))
        (mary (amb 'downing 'hall 'moore 'parker))
        (melissa 'hood)
        (rosalind (amb 'downing 'hall 'moore 'parker)))
    (define (yacht-of father)
      (cond ((eq? father 'downing) melissa)
            ((eq? father 'hall) rosalind)
            ((eq? father 'hood) gabrielle)
            ((eq? father 'moore) lorna)
            (else mary)))
    (require (eq? 'parker (yacht-of gabrielle)))
    (require (distinct? (list gabrielle lorna mary melissa rosalind)))
    (list (list 'gabrielle gabrielle)
          (list 'lorna lorna)
          (list 'mary mary)
          (list 'melissa melissa)
          (list 'rosalind rosalind)))
;;; Starting a new problem
;;; Amb-Eval value:
ok

;;; Amb-Eval input:
(daughters2)
;;; Starting a new problem
;;; Amb-Eval value:
((gabrielle hall) (lorna downing) (mary moore) (melissa hood) (rosalind parker))

;;; Amb-Eval input:
try-again
;;; Amb-Eval value:
((gabrielle hall) (lorna moore) (mary downing) (melissa hood) (rosalind parker))

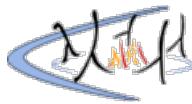
;;; Amb-Eval input:
try-again
;;; Amb-Eval value:
((gabrielle moore) (lorna parker) (mary downing) (melissa hood) (rosalind hall))

;;; Amb-Eval input:
try-again
;;; Amb-Eval value:
((gabrielle moore) (lorna parker) (mary hall) (melissa hood) (rosalind downing))

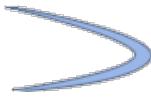
;;; Amb-Eval input:
try-again
;;; There are no more values of
(daughters2)

;;; Amb-Eval input:
End of input stream reached.
Fortitudine vincimus.
```





# sicp-ex-4.44



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.43) | Index | Next exercise (4.45) >>

woofy Short and clean

```
(define (no-conflict col board)
  (define (iter next k)
    (or (null? next)
        (and (not (= (car next) col))
             (not (= (car next) (- col k)))
             (not (= (car next) (+ col k))))))
        (iter (cdr next) (+ k 1))))
  (iter board 1))

(define (queens n)
  (define (iter row result)
    (if (= row n)
        (display result)
        (let ((col (an-integer-between 0 (- n 1))))
          (require (no-conflict col result))
          (iter (+ row 1) (cons col result)))))
  (iter 0 '())))
```

wocanmei

```
;;
;;
;;
;; plain and straightforward solution
(define (queens)
  (let ((q1 (amb 1 2 3 4 5 6 7 8)))
  (let ((q2 (amb 1 2 3 4 5 6 7 8)))
  (require (safe? q2 2 (rows->poses (list q1))))
  (let ((q3 (amb 1 2 3 4 5 6 7 8)))
  (require (safe? q3 3 (rows->poses (list q1 q2))))
  (let ((q4 (amb 1 2 3 4 5 6 7 8)))
  (require (safe? q4 4 (rows->poses (list q1 q2 q3))))
  (let ((q5 (amb 1 2 3 4 5 6 7 8)))
  (require (safe? q5 5 (rows->poses (list q1 q2 q3 q4))))
  (let ((q6 (amb 1 2 3 4 5 6 7 8)))
  (require (safe? q6 6 (rows->poses (list q1 q2 q3 q4 q5))))
  (let ((q7 (amb 1 2 3 4 5 6 7 8)))
  (require (safe? q7 7 (rows->poses (list q1 q2 q3 q4 q5 q6))))
  (let ((q8 (amb 1 2 3 4 5 6 7 8)))
  (require (safe? q8 8 (rows->poses (list q1 q2 q3 q4 q5 q6 q7)))))))
  (rows->poses (list q1 q2 q3 q4 q5 q6 q7 q8)))))))
```

;; helper functions

```
(define (and a b c d)
  (cond ((not a) false)
        ((not b) false)
        ((not c) false)
        ((not d) false)
        (else true)))
```

```
(define (or a b)
  (if a
      true
      b))
```

;; 2.42

```
(define (same-row? p1 p2)
  (= (car p1) (car p2)))

(define (same-col? p1 p2)
  (= (cdr p1) (cdr p2)))

(define (same-diag? p1 p2)
  (let ((row1 (car p1)))
```

```

(coll1 (cdr p1))
(row2 (car p2))
(col2 (cdr p2)))
(or (= (+ row1 coll1) (+ row2 col2))
(= (- row1 coll1) (- row2 col2)))))

(define (safe? row col positions)
(define (safe-iter kp other-positions)
(if (null? other-positions)
true
(and (not (same-row? kp (car other-positions)))
(not (same-col? kp (car other-positions)))
(not (same-diag? kp (car other-positions))))
(safe-iter kp (cdr other-positions))))
(safe-iter (cons row col) positions))

(define (map proc items)
(if (null? items)
'()
(cons (proc (car items))
(map proc (cdr items)))))

(define (rows->poses rows)
(define count 0)
(map (lambda (row)
(begin (set! count (+ count 1))
(cons row count)))
rows))

(queens)

```

xdavidliu

```

;;
;;
;;
(define (vulnerable? queen1-position queen2-position column-separation)
(let ((row-separation (abs (- queen1-position queen2-position))))
(or (= row-separation 0)
(= row-separation column-separation)))))

;; first element of previous-queens is the position of the queen
;; in the column immediately adjacent to next-queen
(define (next-queen-vulnerable? next-queen previous-queens)
(define (iter prev-qs column-separation)
(if (null? prev-qs)
false
(or (vulnerable? next-queen (car prev-qs) column-separation)
(iter (cdr prev-qs) (1+ column-separation)))))

(iter previous-queens 1))

;; use let* even though bindings are independent in order to guarantee efficient nesting
with respect to amb.
(define (eight-queens)
(define (nnqv? next-queen previous-queens)
(not (next-queen-vulnerable? next-queen previous-queens)))
(let* ((prev0 '())
(q1 (amb 1 2 3 4 5 6 7 8)))
(require (nnqv? q1 prev0)) ;; trivially never fails
(let* ((prev1 (cons q1 prev0))
(q2 (amb 1 2 3 4 5 6 7 8)))
(require (nnqv? q2 prev1))
(let* ((prev2 (cons q2 prev1))
(q3 (amb 1 2 3 4 5 6 7 8)))
(require (nnqv? q3 prev2))
(let* ((prev3 (cons q3 prev2))
(q4 (amb 1 2 3 4 5 6 7 8)))
(require (nnqv? q4 prev3))
(let* ((prev4 (cons q4 prev3))
(q5 (amb 1 2 3 4 5 6 7 8)))
(require (nnqv? q5 prev4))
(let* ((prev5 (cons q5 prev4))
(q6 (amb 1 2 3 4 5 6 7 8)))
(require (nnqv? q6 prev5))
(let* ((prev6 (cons q6 prev5))
(q7 (amb 1 2 3 4 5 6 7 8)))
(require (nnqv? q7 prev6))
(let* ((prev7 (cons q7 prev6))
(q8 (amb 1 2 3 4 5 6 7 8)))
(require (nnqv? q8 prev7))
(cons q8 prev7)))))))
;;
use try-again to go through all the solutions.

```

meteorgan

```
; 4.44
(define (enumerate-interval low high)
  (if (> low high)
      '()
      (cons low (enumerate-interval (+ low 1) high)))))

(define (attack? row1 col1 row2 col2)
  (or (= row1 row2)
      (= col1 col2)
      (= (abs (- row1 row2)) (abs (- col1 col2)))))

;; positions is the list of row of former k-1 queens
(define (safe? k positions)
  (let ((kth-row (list-ref positions (- k 1))))
    (define (safe-iter p col)
      (if (>= col k)
          true
          (if (attack? kth-row k (car p) col)
              false
              (safe-iter (cdr p) (+ col 1)))))

(safe-iter positions 1))

(define (list-amb li)
  (if (null? li)
      (amb)
      (amb (car li) (list-amb (cdr li)))))

(define (queens board-size)
  (define (queen-iter k positions)
    (if (= k board-size)
        positions
        (let ((row (list-amb (enumerate-interval 1 board-size))))
          (let ((new-pos (append positions (list row))))
            (require (safe? k new-pos))
            (queen-iter (+ k 1) new-pos)))))

(queen-iter 1 '()))
```

Felix021

```
; a simpler version.

(define (an-integer-between a b)
  (require (<= a b))
  (amb a (an-integer-between (+ a 1) b)))

;;check if (car solution) is compatible with any of (cdr solution)
(define (safe? solution)
  (let ((p (car solution)))
    (define (conflict? q i)
      (or
        (= p q)
        (= p (+ q i))
        (= p (- q i))))
    (define (check rest i)
      (cond
        ((null? rest) #t)
        ((conflict? (car rest) i) #f)
        (else (check (cdr rest) (inc i)))))

    (check (cdr solution) 1)))

(define (queens n)
  (define (iter solution n-left)
    (if (= n-left 0)
        (begin
          (display solution)
          (newline))
        (begin
          (let ((x-solution (cons (an-integer-between 1 n) solution)))
            (require (safe? x-solution))
            (iter x-solution (- n-left 1))))))
    (iter '() n))

(queens 8)
```

donald

```
;use the original method
```

```

(define (enumerate-interval l h)
  (if (> l h)
      '()
      (cons l (enumerate-interval (+ l 1) h))))
(define empty-board '())
(define (adjoin-position row col rest)
  (cons (list row col) rest))
(define (extract item lst)
  (define (scan items)
    (cond ((null? items)
           '())
          ((equal? item (car items))
           (scan (cdr items)))
          (else (cons (car items) (scan (cdr items)))))))
  (scan lst))
(define (safe? col positions)
  (define (iter l)
    (if (null? l)
        true
        (and (car l) (iter (cdr l)))))

(let ((row (caar (filter (lambda (p)
                           (eq? col (cadr p)))
                           positions)))
      (iter (map (lambda (p)
                   (not (or (eq? row (car p))
                             (eq? (- row col) (- (car p) (cadr p)))
                             (eq? (+ row col) (+ (car p) (cadr p)))))))
                  (extract (list row col) positions)))))

(define (queens board-size)
  (define (queen-cols k)
    (if (= k 0)
        (list empty-board)
        (map (lambda (positions)
               (require (safe? k positions))
               (flatmap (lambda (rest-of-queens)
                          (adjoin-position new-row (amb (enumerate-interval 1 board-size))
                                          rest-of-queens))
                         (queen-cols (- k 1))))))
         (queen-cols board-size)))

```

codybartfast

```

(define (new-queen col)
  (cons col (amb 1 2 3 4 5 6 7 8)))

(define (8queens)
  (define (iter queens)
    (require (distinct? (map cdr queens)))
    (require (distinct? (map (lambda (q) (- (car q) (cdr q))) queens)))
    (require (distinct? (map (lambda (q) (+ (car q) (cdr q))) queens)))
    (if (= 8 (length queens))
        queens
        (iter (cons (new-queen (+ 1 (length queens))) queens))))
    (iter '())))
  ;;= ((8 . 4) (7 . 2) (6 . 7) (5 . 3) (4 . 6) (3 . 8) (2 . 5) (1 . 1))

```

revc

I keep track of all the history of my solutions.

```

;; Exercise 4.44
(define (Eight-Queen)
  (define (add-constraint Rx Ry diff)
    (require (not (= Rx Ry)))
    (require (not (= (abs (- Ry Rx)) diff)))))

(let ([r1 (amb 1 2 3 4 5 6 7 8)])
  (let ([r2 (amb 1 2 3 4 5 6 7 8)])
    (add-constraint r1 r2 1)
    (let ([r3 (amb 1 2 3 4 5 6 7 8)])
      (add-constraint r1 r3 2)
      (add-constraint r2 r3 1)
      (let ([r4 (amb 1 2 3 4 5 6 7 8)])
        (add-constraint r1 r4 3)
        (add-constraint r2 r4 2)
        (add-constraint r3 r4 1)
        (let ([r5 (amb 1 2 3 4 5 6 7 8)])
          (add-constraint r1 r5 4)

```

```

(add-constraint r2 r5 3)
(add-constraint r3 r5 2)
(add-constraint r4 r5 1)
(let ([r6 (amb 1 2 3 4 5 6 7 8)])
  (add-constraint r1 r6 5)
  (add-constraint r2 r6 4)
  (add-constraint r3 r6 3)
  (add-constraint r4 r6 2)
  (add-constraint r5 r6 1)
  (let ([r7 (amb 1 2 3 4 5 6 7 8)])
    (add-constraint r1 r7 6)
    (add-constraint r2 r7 5)
    (add-constraint r3 r7 4)
    (add-constraint r4 r7 3)
    (add-constraint r5 r7 2)
    (add-constraint r6 r7 1)
    (let ([r8 (amb 1 2 3 4 5 6 7 8)])
      (add-constraint r1 r8 7)
      (add-constraint r2 r8 6)
      (add-constraint r3 r8 5)
      (add-constraint r4 r8 4)
      (add-constraint r5 r8 3)
      (add-constraint r6 r8 2)
      (add-constraint r7 r8 1)
      (list r1 r2 r3 r4 r5 r6 r7 r8))))))))))

;; Additional Exercise 4.44
(define (Eight-Queen)
  (define (add-constraint Rx Ry diff)
    (require (not (= Rx Ry)))
    (require (not (= (abs (- Ry Rx)) diff)))))

  ;; add constraints to all two elements, both of them are from ``rows``
  (define (add-constraints rows)

    ;; add constraints to all two elements where one is the CAR of ``rows`` and the other
    ;; is from the CDR ``rows``
    (define (loop items diff)
      (if (not (null? items))
          (begin
            (add-constraint (car rows) (car items) diff)
            (loop (cdr items) (+ diff 1)))))

    (if (not (null? (cdr rows)))
        (begin
          (loop (cdr rows) 1)
          (add-constraints (cdr rows)))))

  (let ([r1 (amb 1 2 3 4 5 6 7 8)]
    [r2 (amb 1 2 3 4 5 6 7 8)]
    [r3 (amb 1 2 3 4 5 6 7 8)]
    [r4 (amb 1 2 3 4 5 6 7 8)]
    [r5 (amb 1 2 3 4 5 6 7 8)]
    [r6 (amb 1 2 3 4 5 6 7 8)]
    [r7 (amb 1 2 3 4 5 6 7 8)]
    [r8 (amb 1 2 3 4 5 6 7 8)])
  (add-constraints (list r1 r2 r3 r4 r5 r6 r7 r8))
  (list r1 r2 r3 r4 r5 r6 r7 r8)))))

(define (Eight-Queen)
  ;; add a constraint to two elements with a specified difference
  (define (add-constraint Rx Ry diff)
    (require (not (= Rx Ry))) ; not in a same column
    (require (not (= (abs (- Ry Rx)) diff)))) ; not in a same diagonal

  ;; add constraints to all two elements where one is the CAR of ``rows`` and the other
  ;; is from the CDR ``rows``
  ;; the procedure take a list which is ordered by "descended" (from high row to low row)
  (define (add-constraints rows)
    (define (loop rest diff)
      (if (not (null? rest))
          (begin
            (add-constraint (car rows) (car rest) diff)
            (loop (cdr rest) (+ diff 1)))))

    (loop (cdr rows) 1))

  (let ([r1 (amb 1 2 3 4 5 6 7 8)])
    (let ([r2 (amb 1 2 3 4 5 6 7 8)])
      (add-constraints (list r2 r1))
      (let ([r3 (amb 1 2 3 4 5 6 7 8)])
        (add-constraints (list r3 r2 r1))
        (let ([r4 (amb 1 2 3 4 5 6 7 8)])
          (add-constraints (list r4 r3 r2 r1))
          (let ([r5 (amb 1 2 3 4 5 6 7 8)])
            (add-constraints (list r5 r4 r3 r2 r1))
            (let ([r6 (amb 1 2 3 4 5 6 7 8)])
              (add-constraints (list r6 r5 r4 r3 r2 r1)))))))))))

```

```

        (add-constraints (list r6 r5 r4 r3 r2 r1))
    (let ([r7 (amb 1 2 3 4 5 6 7 8)])
        (add-constraints (list r7 r6 r5 r4 r3 r2 r1))
    (let ([r8 (amb 1 2 3 4 5 6 7 8)])
        (add-constraints (list r8 r7 r6 r5 r4 r3 r2 r1))
    (list r1 r2 r3 r4 r5 r6 r7 r8))))))))))

(define (Eight-Queen)
;; add a constraint to two elements with a specified difference
(define (add-constraint Rx Ry diff)
  (require (not (= Rx Ry))) ; not in a same column
  (require (not (= (abs (- Ry Rx)) diff)))) ; not in a same diagonal

;; add constraints to all two elements where one is the CAR of ``rows`` and the other
;; is from the CDR ``rows``
;; the procedure take a list which is ordered by "descended" (from high row to low row)
(define (add-constraints rows)
  (define (loop rest diff)
    (if (not (null? rest))
        (begin
          (add-constraint (car rows) (car rest) diff)
          (loop (cdr rest) (+ diff 1))))))
  (loop (cdr rows) 1))

(define (get-solution r-rows stage)
  (if (< stage 9)
      (let ([row (amb 1 2 3 4 5 6 7 8)])
        (let ([new-rows (cons row r-rows)])
          (add-constraints new-rows)
          (get-solution new-rows (+ stage 1)))
        (reverse r-rows)))

      (get-solution (list (amb 1 2 3 4 5 6 7 8)) 2))

(define (N-Queen n)
;; add a constraint to two elements with a specified difference
(define (add-constraint Rx Ry diff)
  (require (not (= Rx Ry))) ; not in a same column
  (require (not (= (abs (- Ry Rx)) diff)))) ; not in a same diagonal

;; add constraints to all two elements where one is the CAR of ``rows`` and the other
;; is from the CDR ``rows``
;; the procedure take a list which is ordered by "descended" (from high row to low row)
(define (add-constraints rows)
  (define (loop rest diff)
    (if (not (null? rest))
        (begin
          (add-constraint (car rows) (car rest) diff)
          (loop (cdr rest) (+ diff 1))))))
  (loop (cdr rows) 1))

(define (get-solution r-rows stage)
  (if (< stage n)
      (let ([row (an-integer-between 1 n)])
        (let ([new-rows (cons row r-rows)])
          (add-constraints new-rows)
          (get-solution new-rows (+ stage 1)))
        (reverse r-rows)))

      (get-solution (list (an-integer-between 1 n)) 1))

;; represent Queens as a list of columns in the order of the rows
(define (N-Queen n)
;; check if two queens coexist with each other
(define (offensive? Rx Ry diff)
  (if (= Rx Ry) ; in a same column
      true
      (= (abs (- Ry Rx)) diff))) ; in a same diagonal

;; check if (car solution) is compatible with any of (cdr solution)
(define (safe? rows)
  (define (check rest diff)
    (if (not (null? rest))
        (if (offensive? (car rows) (car rest) diff)
            false
            (check (cdr rest) (+ diff 1)))
        true))
  (check (cdr rows) 1))

(define (queen-iter r-rows stage)
  (if (< stage n)
      (let ([row (an-integer-between 1 n)])
        (let ([new-rows (cons row r-rows)])
          (require (safe? new-rows))
          (queen-iter new-rows (+ stage 1)))
        (reverse r-rows)))

```

```
(queen-iter (list (an-integer-between 1 n)) 1))
```

closeparen

Full brute force using "distinct" was too slow to even attempt. At first I optimized by generating permutations as follows:

```
(define (a-permutation-of s)
  (define (amb-permutations s)
    (define (proc x)
      (map (lambda (p) (cons x p)) (amb-permutations (delete x s)))))

    (define (iter sequence)
      (if (null? sequence)
          (amb)
          (amb (proc (car sequence))
                (iter (cdr sequence))))))

    (if (null? s)
        (list '())
        (iter s)))

  (car (amb-permutations s)))
```

Then ruling out those permutations with diagonal conflicts. That was still pretty slow so I went for a dynamic programming solution:

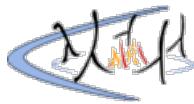
```
(define (n-queens-dp n k)

  (define (choices rest-of-board)
    (define (conflicts-diagonal? x op board)
      (cond ((null? board) false)
            ((= (car board) x) true)
            (else (conflicts-diagonal? (op x) op (cdr board)))))

    (define (good-choice choice)
      (cond ((memq choice rest-of-board) false)
            ((conflicts-diagonal? (- choice 1)
                                  (lambda (x) (- x 1)) rest-of-board) false)
            ((conflicts-diagonal? (+ choice 1)
                                  (lambda (x) (+ x 1)) rest-of-board) false)
            (else true)))

    (filter good-choice (upto k)))

  (if (= n 0)
      '()
      (let ((rest-of-board (n-queens-dp (- n 1) k)))
        (cons (an-element-of (choices rest-of-board))
              rest-of-board))))
```



# sicp-ex-4.45

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.44) | Index | Next exercise (4.46) >>

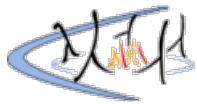
meteorgan

```
;;; Amb-Eval input:  
(parse '(the professor lectures to the student in the class with the cat))  
  
;;; Starting a new problem  
;;; Amb-Eval output:  
(sentence  
  (noun-phrase (articles the) (nouns professor))  
  (verb-phrase (verb-phrase  
    (verb-phrase (verb lectures)  
      (pre-phrase (prep to) (noun-phrase (articles the) (nouns  
student))))  
    (pre-phrase (prep in) (noun-phrase (articles the) (nouns class))))  
  (pre-phrase (prep with) (noun-phrase (articles the) (nouns cat))))  
  
;;; Amb-Eval input:  
try-again  
  
;;; Amb-Eval output:  
(sentence (noun-phrase (articles the) (nouns professor))  
  (verb-phrase (verb-phrase (verb lectures)  
    (pre-phrase (prep to)  
      (noun-phrase (noun-phrase (articles the) (nouns  
class))  
    (pre-phrase (prep with)  
      (noun-phrase (articles the)  
      (nouns cat)))))))  
  
;;; Amb-Eval input:  
try-again  
  
;;; Amb-Eval output:  
(sentence (noun-phrase (articles the) (nouns professor))  
  (verb-phrase (verb lectures)  
    (pre-phrase (prep to)  
      (noun-phrase (noun-phrase (noun-phrase (articles the)  
      (nouns student))  
    (pre-phrase (prep in)  
      (noun-phrase  
      (articles the) (nouns class))))))  
  (pre-phrase (prep with) (noun-phrase (articles the) (nouns  
cat))))  
  
;;; Amb-Eval input:  
try-again  
  
;;; Amb-Eval output:  
(sentence (noun-phrase (articles the) (nouns professor))  
  (verb-phrase (verb lectures)  
    (pre-phrase (prep to)  
      (noun-phrase (noun-phrase (noun-phrase (articles the)  
      (nouns  
student))  
    (pre-phrase (prep in)  
      (noun-phrase  
      (articles the)  
      (nouns class))))))  
  (pre-phrase (prep with) (noun-phrase  
  (articles the) (nouns cat))))))  
  
;;; Amb-Eval input:  
try-again  
  
;;; Amb-Eval output:
```

```
(sentence (noun-phrase (articles the) (nouns professor))
         (verb-phrase (verb lectures)
                      (pre-phrase (prep to)
                                  (noun-phrase (noun-phrase (articles the)
                                              (nouns student)))
                                  (pre-phrase (prep in)
                                              (noun-phrase (noun-phrase
(narticles the)
(nouns class)))
(prep with)
(noun-phrase (articles the)
(nouns cat))))))))
```

---

Last modified : 2012-08-04 01:54:08  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-4.46

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

[<< Previous exercise \(4.45\)](#) | [Index](#) | [Next exercise \(4.47\) >>](#)

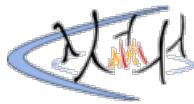
---

meteorgan

That's because function parse-word handles \*unparsed\* from left to right. **If** evaluation has other order, it will conflict with parse-word.

---

Last modified : 2012-08-04 02:16:42  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



[\*\*<< Previous exercise \(4.46\) | Index | Next exercise \(4.48\) >>\*\*](#)

woofy

1. If the input does not start with a verb the process will try to read and trackback on the first word repeatedly forever

2. The procedure will call itself indefinitely when evaluating the argument value and won't even start to read input

In the domain of compiler, the technique in the book maybe corresponds to the left recursion elimination. That is to transform the production (that generate 1 'r' followed by 0 or multiple 'x') :

$A \Rightarrow Ax \mid r$   
(Louis' approach)

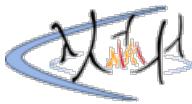
to:

$B \Rightarrow \epsilon \mid xB$   
 $A \Rightarrow rB$   
(Textbook's approach)

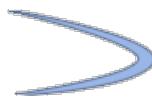
so that there is no production on the right starts with the symbol on the left (left recursion). The effect is that if we follow the rules directly we won't stuck in infinite recursion (we have choices to stop or to go deeper).

meteorgan

This doesn't work. Because the second branch of amb expression will call (parse-verb-phrase) again, this will lead to infinite loop. **If** we change the order in amb, it still will lead to infinite loop.



# sicp-ex-4.48



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
 [Edit] [Edit History]  
 Search:

<< Previous exercise (4.47) | Index | Next exercise (4.49) >>

xdavidliu

First, let's implement adjectives. Unlike noun-phrases or verb-phrases, there isn't the possibility of any fancy recursive structures. For example, in the phrase "a delicious pig in a blanket", there is no difference in meaning between "(a delicious pig) in a blanket" or "a delicious (pig in a blanket)". This is because the phrase "in a blanket" effectively \*acts\* as an adjective, so it really doesn't matter the order in which we "attach" adjectives to the noun "pig".

Hence, we can assume that adjectives only affect simple nouns:

```
(define adjectives '(adjective big small green round))

;; replaces definition from book
(define (parse-simple-noun-phrase)
  (let ((art (parse-word articles))
        (tag 'simple-noun-phrase))
    (amb (list tag art (parse-word adjectives) (parse-word nouns))
         (list tag art (parse-word nouns)))))
```

We can make the same assumption for adverbs, as long as we assume adverbs always \*precede\* verbs, and don't allow fancy sentences like "the boy sees through a glass, darkly".

```
(define adverbs '(adverb quickly slowly lazily eagerly))

;; new function, not in book
(define (parse-simple-verb)
  (amb (list 'simple-verb
             (parse-word adverbs)
             (parse-word verbs))
       (parse-word verbs)))

;; replaces definition from book
;; only difference is the call to parse-simple-verb
(define (parse-verb-phrase)
  (define (maybe-extend verb-phrase)
    (amb verb-phrase
         (maybe-extend (list 'verb-phrase
                            verb-phrase
                            (parse-prepositional-phrase))))))
  (maybe-extend (parse-simple-verb)))
```

Compound sentences such as "the man eats the dinner when the sun sets", on the other hand, \*are\* recursive structures, and just so happen to have the same recursiveness as verb-phrases and noun-phrases. Suppose A, B, and C are non-compound sentences, and \*unparsed\* contains them in that order, separated by conjunctions like "while", "and", "or", "before", etc. Then parsing successfully would require outputting all possible "parenthesizations": A(BC) and (AB)C. Inputs with larger numbers of non-compound sentences separated by conjunctions, like ABCDEF..., have even more possible parenthesizations. The structure of this problem is similar to that of parenthesizations of matrix multiplications, as discussed in one of the early chapters of CLRS.

Fortunately, the maybe-extend functions from the book makes enumerating all possible parenthesizations of compound sentences very easy and elegant.

```
(define conjunctions '(conjunction while when but and or))

;; new function not included in book, but this is just a slight modification of the
;; original parse-sentence
(define (parse-simple-sentence)
  (list 'simple-sentence
        (parse-noun-phrase)
        (parse-verb-phrase)))

;; replaces definition from book
(define (parse-sentence)
  (define (maybe-extend sentence)
    (amb sentence
         (maybe-extend (list 'compound-sentence
                            sentence
                            (parse-word conjunctions)
                            (parse-sentence))))))
```

```
(maybe-extend (parse-simple-sentence)))
```

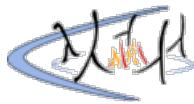
Note the reason that maybe-extend can be copied word-for-word without any modification is that the structure of compound-sentences "linked" by conjunctions is exactly the same as noun-phrases linked by prepositions.

meteorgan

```
(define adjectives '(adjective ugly stupid lazy dirty shitty))
(define (parse-simple-noun-phrase)
  (amb (list 'simple-noun-phrase
             (parse-word articles)
             (parse-word nouns))
        (list 'simple-noun-phrase
              (parse-word articles)
              (parse-word adjectives)
              (parse-word nouns))))
```

---

Last modified : 2018-06-12 21:11:08  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-4.49

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.48) | Index | Next exercise (4.50) >>

meteorgan

```
(define (list-amb li)
  (if (null? li)
      (amb)
      (amb (car li) (list-amb (cdr li)))))

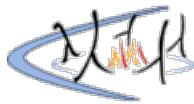
(define (parse-word word-list)
  (require (not (null? *unparsed*)))
  (require (memq (car *unparsed*) (cdr word-list)))
  (let ((found-word (car *unparsed*)))
    (set! *unparsed* (cdr *unparsed*))
    (list-amb (cdr word-list))) ;; change
```

gets:

the student for the student studies for the student  
the student for the student studie for the professor  
the student for the student studie for the cat  
the student for the student studie for the class  
the student for the student studie for a student  
the student for the student studie for a professor  
the student for the student studie for a cat  
the student for the student studie for a class  
the student for the student studie to the student  
the student for the student studie to the professor

donald

```
(define (parse-word2 word-list)
  (require (not (null? *unparsed*)))
  (set! *unparsed* (cdr *unparsed*))
  (list (car word-list) (amb (cdr word-list))))
```



# sicp-ex-4.50



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.49) | Index | Next exercise (4.51) >>

woofy

True shuffling.

```
(load "evaluator-amb.scm")

(define (shuffled s)
  (define (swap s p q)
    (let ((ps (list-starting-from s p))
          (qs (list-starting-from s q)))
      (let ((pv (car ps)))
        (set-car! ps (car qs))
        (set-car! qs pv))))
  (define (iter rest)
    (if (null? rest)
        s
        (let ((n (random (length rest))))
          (swap rest 0 n)
          (iter (cdr rest))))))
  (iter s))

(define (analyze-amb exp)
  (let ((cprocs (map analyze (amb-choices exp)))) ; can't shuffle here
    (lambda (env succeed fail)
      ; achieve random order by shuffling choices at RUNTIME
      (define shuffled-cprocs (shuffled cprocs))
      (define (try-next choices)
        (if (null? choices)
            (fail)
            ((car choices) env
             succeed
             (lambda ()
               (try-next (cdr choices))))))
      (try-next shuffled-cprocs)))
```

meteorgan

```
; In (analyze expr) adds
((ramb? expr) (analyze-ramb expr))

;; add these code to amb evaluator
(define (analyze-ramb expr)
  (analyze-amb (cons 'amb (ramb-choices expr)))))

;; amb expression
(define (amb? expr) (tagged-list? expr 'amb))
(define (amb-choices expr) (cdr expr))

(define (ramb? expr) (tagged-list? expr 'ramb))
(define (ramb-choices expr) (shuffle-list (cdr expr)))

;; random-in-place, from CLRS 5.3
(define (shuffle-list lst)
  (define (random-shuffle result rest)
    (if (null? rest)
        result
        (let* ((pos (random (length rest)))
               (item (list-ref rest pos)))
          (if (= pos 0)
              (random-shuffle (append result (list item)) (cdr rest))
              (let ((first-item (car rest)))
                (random-shuffle (append result (list item))
                               (insert! first-item (- pos 1) (cdr (delete! pos
rest))))))))))
```

```

(random-shuffle '() lst))

;; insert item to lst in position k.
(define (insert! item k lst)
  (if (or (= k 0) (null? lst))
      (append (list item) lst)
      (cons (car lst) (insert! item (- k 1) (cdr lst)))))
(define (delete! k lst)
  (cond ((null? lst) '())
        ((= k 0) (cdr lst))
        (else (cons (car lst)
                     (delete! (- k 1) (cdr lst))))))

```

Rptx

```

; this procedure gets the random element to the start of the list. The rest
; is the same as in amb.

(define (analyze-ramb exp)
  (define (list-ref-and-delete ref lst) ; get random item from list.
    (define (loop count prev-items rest-items) ; and return a list with the
      (if (= count 0) ; random item as its car
          (cons (car rest-items) ; and the rest of the list as the cdr
                (append prev-items (cdr rest-items)))
          (loop (- count 1) ; this will mangle the list every time
                (cons (car rest-items) ; creating a "random" amb.
                      prev-items)
                (cdr rest-items))))
    (if (null? lst)
        '()
        (loop ref '() lst)))
  (let ((cprocs (map analyze (amb-choices exp))))
    (lambda (env succeed fail)
      (define (try-next choices)
        (if (null? choices)
            (fail)
            (let ((randomized (list-ref-and-delete
                               (random (length choices))
                               choices)))
              ((car randomized) env
               succeed
               (lambda ()
                 (try-next (cdr randomized)))))))
      (try-next cprocs)))

```

poly

This solution is kind of like the above one, but with a clear process.

```

;; version 1
(define (analyze-ramb exp)
  (let ((cprocs (map analyze (amb-choices exp))))
    (lambda (env succeed fail)
      (define (try-next choices)
        (if (null? choices)
            (fail)
            (let ((l (length choices)))
              (let ((c (list-ref choices (random l))))
                (c env
                  succeed
                  (lambda ()
                    (try-next (remove choices c)))))))
      (try-next cprocs)))

(define (remove seq elt)
  (filter (lambda (x) (not (eq? elt x))) seq))

;; version 2: won't do extra works during running
(define (shuffle seq)
  (define (iter seq res)
    (if (null? seq)
        res
        (let ((index (random (length seq))))
          (let ((element (list-ref seq index)))
            (iter (remove seq element)
                  (cons element res))))))
  (iter seq nil))

(define (ramb-choices exp) (shuffle (cdr exp)))

; analyze-ramb is the same as analyze-amb

```

aos

A simple `shuffle-list` can be used once the choices are gotten and then we can apply this method to our choices:

```
(define (shuffle lst)
  (map cdr
    (sort
      (map (lambda (x) (cons (random 1.0) x)) lst)
      (lambda (x y) (< (car x) (car y))))))

(define (analyze-amb exp)
  (let ((cprocs
        (map analyze (amb-choices exp))))
    (lambda (env succeed fail)
      (define (try-next choices)
        (if (null? choices)
            (fail)
            ((car choices)
             env
             succeed
             (lambda ()
               (try-next (cdr choices)))))))
      (try-next (shuffle cprocs)))) ;; here -- or basically anywhere where we grab the
                                     choices
```

revc

The preceding solutions are either shuffling choices during analysis or choosing a random choice in running time. Both methods have flaws, which do not meet the needs of Exercise.

The first method just randomly interprets ramb expressions by rearranging the order of choices, if we define a function with ramb, no matter how many times we call it, its behavior doesn't change, because the order of choices has been fixed during analysis, even if it is random.

The second method try to select a choice in running time randomly, but it makes a mistake that is it treats every option equally. For amb expression having recursion, there needs different weights to make a choice. For instance, ``an-integer-between`` with the second method have 50% chance to select ``low`` which results in a problem that the function prefers to return the numbers At the beginning.

My solution tackles this problem with a technique—specify the probability of a choice. See the following code:

```
;;; Exercise 4.50

(define the-default-succeed (lambda (value fail) value))
(define the-default-fail (lambda () 'fail))

(define (analyze exp)
  (cond ((self-evaluating? exp)
         (analyze-self-evaluating exp))
        ((quoted? exp) (analyze-quoted exp))
        ((variable? exp) (analyze-variable exp))
        ((assignment? exp) (analyze-assignment exp))
        ((definition? exp) (analyze-definition exp))
        ((if? exp) (analyze-if exp))
        ((lambda? exp) (analyze-lambda exp))
        ((begin? exp) (analyze-sequence (begin-actions exp)))
        ((cond? exp) (analyze (cond->if exp)))
        ((let? exp) (analyze (let->combination exp))) ;**
        ((amb? exp) (analyze-amb exp)) ;**
        ((ramb? exp) (analyze-ramb exp)) ;**
        ((application? exp) (analyze-application exp))
        (else
         (error "Unknown expression type -- ANALYZE" exp)))

;;; Extended syntax for ``ramb`` is as follows:
;;; (ramb expr ...)
;;; expr ::= value-expr
;;;       || (<Prob> value-expr prob-expr)
;;; NOTE: The probability here tends to be a proportion of likelihood.
;;; The first type of expr has ZERO probability.
;;; The probability of the second type is the value of prob-expr.
;;; The total probability is the sum of the probabilities of all expr.

(define (ramb? exp) (tagged-list? exp 'ramb))
(define (ramb-choices exp) (map
                           (lambda (x)
                             (if (and (pair? x) (eq? (car x) '<Prob>)) (cadr x) x))
                           (cdr exp)))

(define (ramb-probabilities exp) (map
```

```

        (lambda (x)
          (if (and (pair? x) (eq? (car x) '<Prob>)) (caddr x)
0))
          (cdr exp)))

;;; pmf: Probability Mass Function
;;; return a list of pairs consisting of a choice(variable) and a probability.
(define (ramb-pmf exp) (map list (map analyze (ramb-choices exp)) (map analyze (ramb-probabilities exp))))
(define (pmf-variable pair) (car pair))
(define (pmf-probability pair) (cadr pair))

(define (cdf-variable pair) (car pair))
(define (cdf-probability pair) (cadr pair))

;;; Cumulative Distribution Function
;;; return a list of pairs consisting of choice(variable) and a cumulative probability.
(define (CDF pmf)
  (let loop ((cumulation 0)
            (choices-probs pmf)
            (ans '()))
    (if (null? choices-probs)
        (reverse ans)
        (let ((new-cumulation (+ cumulation (pmf-probability (car choices-probs)))))
          (loop new-cumulation
                (cdr choices-probs)
                (cons (list (pmf-variable (car choices-probs)) new-cumulation) ans))))))

;;; sort pmf with probability in descending order, and return its CDF
(define (distribution pmf)
  (let* ((sorted-pmf (sort (lambda (x y) (> (cadr x) (cadr y))) pmf))
         (cdf (CDF sorted-pmf)))
    cdf))

;;; select the first variable if its cumulative probability > r
(define (select-with-random cdf r)
  (if (< r (cdf-probability (car cdf)))
      (car cdf)
      (select-with-random (cdr cdf) r)))

;;; if the total probability n is ZERO, then select a variable with the same possibility,
;;; otherwise generate a number between 0 and n - 1 and then call select-with-random.
(define (select cdf n)
  (if (= n 0)
      (list-ref cdf (random (length cdf)))
      (select-with-random cdf (random n)))))

(define (remove-choice choice pmf)
  (define (loop pair)
    (if (eq? choice (caar pair))
        (remove (car pair) pmf)
        (loop (cdr pair))))
  (loop pmf))

(define (analyze-ramb exp)
  (let ((pmf (ramb-pmf exp)))
    (lambda (env succeed fail)
      (define (try-next pmf)
        (let* ((probs (map (lambda (x) ((pmf-probability x) env the-default-succeed the-default-fail))
                           pmf))
               (total-prob (apply-in-underlying-scheme + probs))
               (new-pmf (map (lambda (x y) (list (car x) y)) pmf probs))
               (cdf (distribution new-pmf)))
          (if (null? pmf)
              (fail)
              (let ((choice (car (select cdf total-prob)))) ; select a choice in running
rather analyzing.
                (choice env
                      succeed
                      (lambda ()
                        (try-next (remove-choice choice pmf))))))))
      (try-next pmf)))))

(ambeval '(define (require p)
            (if (not p) (ramb)))
the-global-environment
the-default-succeed the-default-fail)

(ambeval '(define (an-integer-between low high)
            (require (<= low high))
            (ramb (<Prob> low 1) (<Prob> (an-integer-between (+ low 1) high) (- high
low))))
the-global-environment
the-default-succeed the-default-fail)

;;;;;;;;;;

```

```

;;;; test;;;;;
;;;;;;
;;; Amb-Eval input:
(an-integer-between 1 10)

;;; Starting a new problem
;;; Amb-Eval value:
4

;;; Amb-Eval input:
(an-integer-between 1 10)

;;; Starting a new problem
;;; Amb-Eval value:
2

;;; Amb-Eval input:
(an-integer-between 1 10)

;;; Starting a new problem
;;; Amb-Eval value:
10

;;; Amb-Eval input:
(an-integer-between 1 10)

;;; Starting a new problem
;;; Amb-Eval value:
2

;;; Amb-Eval input:
(an-integer-between 1 10)

;;; Starting a new problem
;;; Amb-Eval value:
8

;;;;;;
;;;help with Alyssa's problem;;
;;;;;

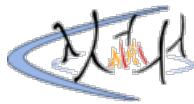
(define (an-element-of items)
  (require (not (null? items)))
  (rmb (car items) (an-element-of (cdr items)))))

(define (generate-word word-list)
  (require (not (null? *unparsed*)))
  (let ((word (an-element-of (cdr word-list))))
    (set! *unparsed* (cdr *unparsed*))
    (list (car word-list) word)))

```

woofy

The shuffle problem you stated could easily be solved by moving the shuffle process into the execution function (shuffle at runtime).



# sicp-ex-4.51

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.50) | Index | Next exercise (4.52) >>

meteorgan

```
; ; In analyze adds
((pernament-set? expr) (analyze-pernament-set expr))

; ; add those code.
(define (pernament-set? expr) (tagged-list? expr 'pernament-set!))

(define (analyze-pernament-set expr)
  (let ((var (assignment-variable expr))
        (vproc (analyze (assignment-value expr))))
    (lambda (env succeed fail)
      (vproc env
        (lambda (val fail2)
          (set-variable-value! var val env)
          (succeed 'ok fail2))
        fail)))))

if use set!, the result will be:
(a b 1) (a c 1) ...
```

Last modified : 2012-08-04 08:03:22  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-4.52

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.51) | Index | Next exercise (4.53) >>

woofy

to summarize

```
(load "evaluator-amb.scm")

(define (if-fail? exp)
  (tagged-list? exp 'if-fail))
(define (if-fail-cond exp) (cadr exp))
(define (if-fail-alt exp) (caddr exp))

(define (analyze-if-fail exp)
  (let ((cproc (analyze (if-fail-cond exp)))
        (aproc (analyze (if-fail-alt exp))))
    (lambda (env succeed fail)
      (cproc env
             succeed
             (lambda () 
               (aproc env succeed fail))))))
```

meteorgan

```
; add this in analyze
((if-fail? expr) (analyze-if-fail expr))

; add those to amb evaluator
(define (if-fail? expr) (tagged-list? expr 'if-fail))

(define (analyze-if-fail expr)
  (let ((first (analyze (cadr expr)))
        (second (analyze (caddr expr))))
    (lambda (env succeed fail)
      (first env
             (lambda (value fail2)
               (succeed value fail))
             (lambda ()
               (second env succeed fail))))))
```

timothy235

The success continuation needs to be (succeed value fail2) and not (succeed value fail). Otherwise, try-again will not return new values even when there are some left.

xavidiliu

in other words,

```
; ;
;
(lambda (value fail2)
  (succeed value fail))
```

should just be changed to succeed.

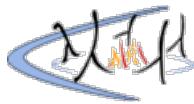
codybartfast

```
(define try-expr cadr)
(define fail-expr caddr)
```

```
(define (analyze-if-fail exp)
  (let ((tproc (analyze (try-expr exp)))
        (fproc (analyze (fail-expr exp))))
    (lambda (env succeed fail)
      (tproc env succeed (lambda () (fproc env succeed fail))))))
```

---

Last modified : 2020-05-05 13:45:43  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-4.53

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.52) | Index | Next exercise (4.54) >>

meteorgan

((8 35) (3 110) (3 20))

uuu

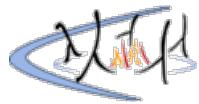
the flow of how we got this result:

```
"prime-sum-pair"-->(3 20)
-->"permanent-set"-->pairs:((3 20))-->"(amb)", which means fail-->another try
-->"prime-sum-pair"-->(3 110)
-->"permanent-set"-->pairs:((3 110) (3 20))-->"(amb)", which means fail-->another try
-->"prime-sum-pair"-->(8 35)
-->"permanent-set"-->pairs:((8 35) (3 110) (3 20))-->"(amb)", which means fail-->another try
-->"prime-sum-pair"-->no more value, fail
-->"if-fail", the <fail> part-->return "pairs"
```

so, if we input "try-again", we will get

"There are no more values of (let ((pairs (quote))) (if-fail ...)"

Last modified : 2017-01-27 03:59:58  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-4.54

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

<< Previous exercise (4.53) | Index | Next exercise (4.55) >>

---

meteorgan

```
(if (not (true? pred-value))
  (fail2)
  (succeed 'ok fail2))
```

woofy

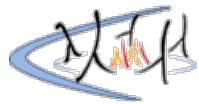
Voted. However it is interesting to think about expressions like

```
(require (amb true false))
```

The simple one above seems simply has the effect as a no-op.

---

Last modified : 2020-05-05 14:13:10  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-4.55

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

[<< Previous exercise \(4.54\)](#) | [Index](#) | [Next exercise \(4.56\) >>](#)

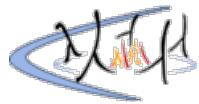
---

meteorgan

- (a) (supervisor ?person (Bitdiddle Ben))
- (b) (job ?person (accounting . ?work))
- (c) (address ?person (Slumerville . ?address))

---

Last modified : 2012-08-04 12:51:04  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-4.56

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

[<< Previous exercise \(4.55\)](#) | [Index](#) | [Next exercise \(4.57\) >>](#)

---

meteorgan

```
(a) (and (supervisor ?person (Bitdiddle Ben))
      (address ?person ?where))
(b) (and (salary (Bitdiddle Ben) ?number)
      (salary ?person ?amount)
      (lisp-value < ?amount ?number)))
(c) (and (supervisor ?person ?boss)
      (not (job ?boss (computer . ?type)))
      (job ?boss ?job)))
```

---

Last modified : 2012-08-04 13:08:18  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-4.57

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.56) | Index | Next exercise (4.58) >>

woofy

one not using positive same:

```
(rule (can-replace ?person1 ?person2)
      (and (job ?person1 ?job1)
              (or (job ?person2 ?job1)
                  (and (job ?person2 ?job2)
                          (can-do-job ?job1 ?job2)))
                  (not (same ?person1 ?person2))))
      ; a
      (can-replace ?x (Fect Cy D))

      ; b
      (and (can-replace ?a ?b)
              (salary ?a ?as)
              (salary ?b ?bs)
              (lisp-value < ?as ?bs))
```

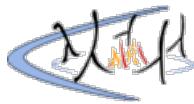
meteorgan

```
(a) (assert! (rule (replace ?person1 ?person2)
                     (and (job ?person1 ?job1)
                            (job ?person2 ?job2)
                            (or (same ?job1 ?job2)
                                  (can-do-job ?job1 ?job2))
                            (not (same ?person1 ?person2)))))

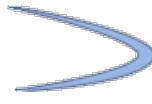
(b)
;; Query input:
(replace ?p (Fect Cy D))

;; Query output:
(replace (Bitdiddle Ben) (Fect Cy D))
(replace (Hacker Alyssa P) (Fect Cy D))

(c)
(and (salary ?p1 ?a1)
        (salary ?p2 ?a2)
        (replace ?p1 ?p2)
        (lisp-value > ?a2 ?a1))
;; Query output:
(and (salary (Aull DeWitt) 25000) (salary (Warbucks Oliver) 150000) (replace (Aull DeWitt)
(Warbucks Oliver)) (lisp-value > 150000 25000))
(and (salary (Fect Cy D) 35000) (salary (Hacker Alyssa P) 40000) (replace (Fect Cy D)
(Hacker Alyssa P)) (lisp-value > 40000 35000))
```



# sicp-ex-4.58



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.57) | Index | Next exercise (4.59) >>

meteorgan

```
rule:  
(assert! (rule (bigshot ?person ?division)  
              (and (job ?person (?division . ?rest))  
                    (or (not (supervisor ?person ?boss))  
                         (and (supervisor ?person ?boss)  
                               (not (job ?boss (?division . ?r)))))))  
;;; Query output:  
(bigshot (Warbucks Oliver) administration)  
(bigshot (Scrooge Eben) accounting)  
(bigshot (Bitdiddle Ben) computer)
```

carpdiem

The bigshot rule should include a recursive call back to itself, as even if a person's direct supervisor isn't in their department, it's possible for their supervisor's supervisor (or so forth) to be in their department, thus disqualifying the original individual from being a bigshot.

e.g.,

```
(rule (bigshot ?person ?division)  
      (and (job ?person (?division . ?rest))  
            (or (not (supervisor ?person ?boss))  
                 (and (supervisor ?person ?boss)  
                       (not (job ?boss (?division . ?r))))  
                 (not (bigshot ?boss ?division))))))
```

imelendez

Great catch! I certainly didn't think to account for this situation. Methinks this could happen with temporary assignments to different departments for a project!

The rule still appears to fail though. Consider that we match on ?boss that isn't in the ?division. The bigshot rule will always fail in that case for ?boss because ?boss isn't in the department (despite ?boss having ?bosses-boss in ?department). We never even checked because it 'short-circuits'. It would appear that it is necessary to have a separate rule that will check if 'any supervisor is in the division' that won't fail if the ?person isn't in the division.

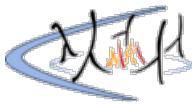
```
(rule (any-supervisor-in-division ?person ?division)  
      ;; recursively find out if a supervisor is in the division  
      ;;  
      ;; this is necessary because the big-shot rule 'short circuits' if  
      ;; ?person is not in ?division, but we are still asking the question  
      ;; even if the current person is not in ?division  
      (and (supervisor ?person ?supervisor)  
            (or (job ?supervisor (?division . ?type))  
                 (any-supervisor-in-division ?supervisor ?division))))  
  
(rule (big-shot ?person ?division)  
      (and (job ?person (?division . ?type))  
            (not (any-supervisor-in-division ?person ?division))))
```

codybartfast

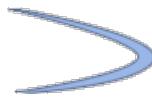
```
(rule (bigshot ?bshot ?division)
```

```
(and (job ?bshot (?division . ?bshot-rest))
     (not (and (supervisor ?bshot ?boss)
                (job ?boss (?division . ?boss-rest))))))

;;; Query results:
(bigshot (Scrooge Eben) accounting)
(bigshot (Warbucks Oliver) administration)
(bigshot (Bitdiddle Ben) computer)
```



# sicp-ex-4.59



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.58) | Index | Next exercise (4.60) >>

meteorgan

```
(a) (meeting ?dept (Friday . ?t))  
(b)  
(rule (meeting-time ?person ?day-and-time)  
      (and (job ?person (?dept . ?r))  
            (or (meeting ?dept ?day-and-time)  
                 (meeting the-whole-company ?day-and-time))))  
(c)  
(and (meeting-time (Hacker Alyssa P) (Wednesday . ?time))  
      (meeting ?dept (Wednesday . ?time))))
```

djrochford

meteorgan's answers look correct to me, but c) is strange. I don't think there is any reason for the second conjunct in the `and` statement. The first should return all meetings Alyssa needs to attend -- that's the point of the rule. And, indeed, it will, on meteorgan's definition of the rule.

alan

According to the examples in the book, I think that rule's <body> will not be printed.

egbc

To expand on this, the first clause will filter out all of the times that Alyssa has a meeting; the second clause will make it so that the query will select the department and time of the meeting. At least, that's my understanding - I find the syntax of this query language to be confusing and unintuitive so I definitely could be wrong...

Here's a version that I think should return all of the info about each meeting Alyssa has on Wednesdays:

(and

```
(meeting-time (Alyssa P Hacker) (Wednesday ?time))  
(meeting . ?meeting-details))
```

donald

```
(b)  
(rule (meeting-time ?p ?day-and-time)  
      (or (meeting the-whole-company ?day-and-time)  
           (and (job ?p (?d . ?rest))  
                 (meeting ?d ?day-and-time))))
```

aos

I'm not sure these are right. The rule states:

"...person's meetings include all whole-company meetings **plus** all meetings of that person's division"

We want ALL meetings, not either "whole-company" meetings or "division" meetings. By removing the `or`, we get:

```
(rule (meeting-time ?person ?day-and-time)  
      (and (meeting whole-company ?day-and-time)  
            (job ?person (?division . ?r))  
            (meeting ?division ?day-and-time))))
```

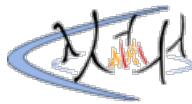
carpdiem

By removing the `(or )` clause, your rule will only match both meetings that match the 'whole-company' and '?division' simultaneously, and since 'whole-

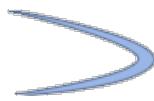
'company' isn't one of the matches for '?division', the query will return no results.

Instead, the (or ) clause allows a match on either meetings that are for 'whole-company' or match '?division', which is the entire, correct list.

In terms of venn-diagrams, the (or ) clause gives us the union of the two sets (meetings with 'whole-company' and meetings with '?division'); while only having an (and ) clause gives us the intersection (empty!) of the same two sets.



# sicp-ex-4.60



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.59) | Index | Next exercise (4.61) >>

woofy

A version that *actually* works with the book's evaluator implementation:

```
(assert! (rule (before ?x ?y)
    (lisp-value
        (lambda (s1 s2)
            (define (list->string s)
                (fold-right
                    string-append
                    ""))
            (map symbol->string s)))
        (string<? (list->string s1) (list->string s2)))
    ?x
    ?y)))

(assert! (rule (lives-near ?person-1 ?person-2)
    (and (address ?person-1 (?town . ?rest-1))
        (address ?person-2 (?town . ?rest-2))
        (before ?person-1 ?person-2)))

; Tests:

;;; Query input:
(lives-near ?x ?y)

;;; Query results:
(lives-near (aull dewitt) (reasoner louis))
(lives-near (aull dewitt) (bitdiddle ben))
(lives-near (fect cy d) (hacker alyssa p))
(lives-near (bitdiddle ben) (reasoner louis))
```

datasnake

The problem with this approach is that it changes the rule to say that `(lives-near ?x ?y)` is only true if they live in the same town and the first name comes before the second alphabetically, which can cause problems with queries where only one name is left unspecified. As an example, take this query:

```
;;; Query input:
(lives-near ?person (Bitdiddle Ben))

;;; Query results:
(lives-near (Aull DeWitt) (Bitdiddle Ben))
(lives-near (Reasoner Louis) (Bitdiddle Ben))
```

Changing the definition of `lives-near` will change the result to the following:

```
;;; Query input:
(lives-near ?person (Bitdiddle Ben))

;;; Query results:
(lives-near (Aull DeWitt) (Bitdiddle Ben))
```

Since "Reasoner" comes after "Bitdiddle" alphabetically, the second result no longer satisfies the rule.

meteorgan

because all the answers satisfy the rule.  
we can sort the person in alphabetic order, then get only one pair.

```
(define (person->string person)
  (if (null? person)
      ""))
  
```

```

(string-append (symbol->string (car person)) (person->string (cdr person))))
(define (person? p1 p2)
  (string>? (person->string p1) (person->string p2)))

(assert! (rule (asy-lives-near ?person1 ?person2)
    (and (address ?person1 (?town . ?rest-1))
        (address ?person2 (?town . ?rest-2))
        (lisp-value person>? ?person1 ?person2))))
```

yd

It seems that you cannot have a "lives-near" function that satisfy both: 1. doesn't print both pair (x y) and (y x) 2.worked both with query (x ?any) and (?any x)

Here is the reason: when 1. is preserved (e.g. (x y) is chosen), that means the other one (y x) doesn't satisfy the query, thus one of (x ?any) and (?any x) will be dropped.

when 2 is preserved, there is no reason for you to choose which one to drop in (x y) and (y x), even you cannot find they are the same one (in terms of meaning), unless using some built-in method to keep saving pairs that have been found during the evaluating.

in general case, to implement this feature, we need some kind of method that find a group of sentences that are the same in terms of meaning. It could be very hard because it's almost like the provable problem in pure math.

SteeleDynamics

It appears to me that there is a much simpler way to implement this: by constructing a predicate procedure and passing it as the predicate argument to a 'lisp-value' query. You can have alphabetical ordering by surname as a constraint, which will eliminate "commutative" frames. Below is the query evaluator input and output:

```

1 |=> (initialize-data-base microshaft-data-base)
;Value: done

1 |=> (query-driver-loop)

;; Query input:
(lives-near ?person (Hacker Alyssa P))
;; Query results:
(lives-near (fect cy d) (hacker alyssa p))

;; Query input:
(lives-near ?person-1 ?person-2)
;; Query results:
(lives-near (aull dewitt) (reasoner louis))
(lives-near (aull dewitt) (bitdiddle ben))
(lives-near (reasoner louis) (aull dewitt))
(lives-near (reasoner louis) (bitdiddle ben))
(lives-near (hacker alyssa p) (fect cy d))
(lives-near (fect cy d) (hacker alyssa p))
(lives-near (bitdiddle ben) (aull dewitt))
(lives-near (bitdiddle ben) (reasoner louis))

;; Query input:
(and (lives-near ?person-1 ?person-2)
      (lisp-value (lambda (a b) (symbol<? (car a) (car b)))
                  ?person-1
                  ?person-2))
;; Query results:
(and (lives-near (aull dewitt) (reasoner louis)) (lisp-value (lambda (a b) (symbol<? (car a) (car b))) (aull dewitt) (reasoner louis)))
      (and (lives-near (aull dewitt) (bitdiddle ben)) (lisp-value (lambda (a b) (symbol<? (car a) (car b))) (aull dewitt) (bitdiddle ben)))
          (and (lives-near (fect cy d) (hacker alyssa p)) (lisp-value (lambda (a b) (symbol<? (car a) (car b))) (fect cy d) (hacker alyssa p)))
              (and (lives-near (bitdiddle ben) (reasoner louis)) (lisp-value (lambda (a b) (symbol<? (car a) (car b))) (bitdiddle ben) (reasoner louis))))))

;; Query input:
End of input stream reached.
Post proelium, praemium.
```

egbc

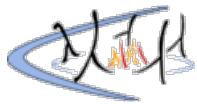
I haven't implemented the query language yet, so not sure if this actually does what I think it does, but I believe this re-implementation of lives-near should work for all possible inputs:

```

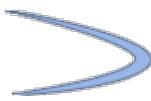
(rule (lives-near ?person-1 ?person-2)
  (or
    (and
      (lisp-value person-alphabetical-lt? ?person-2 ?person-1)
```

```
(lives-near ?person-2 ?person-1))
(and (address ?person-1 (?town . ?rest-1))
      (address ?person-2 (?town . ?rest-2)))
      (not (same ?person-1 ?person-2))))
```

Assuming that person-alphabetical-lt? is some comparison method that does exactly what it sounds like it should.



# sicp-ex-4.61



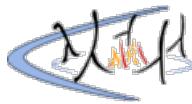
[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.60) | Index | Next exercise (4.62) >>

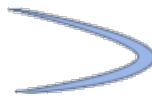
meteorgan

```
;;; Query input:  
(?x next-to ?y in (1 (2 3) 4))  
  
;;; Query output:  
((2 3) next-to 4 in (1 (2 3) 4))  
(1 next-to (2 3) in (1 (2 3) 4))  
  
;;; Query input:  
(?x next-to 1 in (2 1 3 1))  
  
;;; Query output:  
(3 next-to 1 in (2 1 3 1))  
(2 next-to 1 in (2 1 3 1))
```

Last modified : 2012-08-06 13:04:45  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-4.62



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.61) | Index | Next exercise (4.63) >>

meteorgan

```
rule:  
(assert! (rule (last-pair (?x) (?x))))  
(assert! (rule (last-pair (?u . ?v) (?x))  
                (last-pair ?v (?x))))  
  
;;; Query input:  
(last-pair (3) ?x)  
  
;;; Query output:  
(last-pair (3) (3))  
;;; Query input:  
(last-pair (1 2 3) ?x)  
  
;;; Query output:  
(last-pair (1 2 3) (3))  
;;; Query input:  
(last-pair (2 ?x) (3))  
  
;;; Query output:  
(last-pair (2 3) (3))  
  
there is no answer for (last-pair ?x (3))
```

codybartfast

After reversing the two parts of the rule it does return results for the last query.

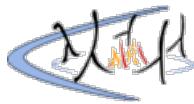
```
Last-Pair Rule (parts reversed):  
=====  
  
(rule (last-pair (?u . ?v) (?x))  
      (last-pair ?v (?x)))  
  
(rule (last-pair (?x) (?x)))  
  
Results (Rules Reversed):  
=====  
  
;;; Query input:  
(last-pair (3) ?x)  
  
;;; Query results:  
(last-pair (3) (3))  
  
;;; Query input:  
(last-pair (1 2 3) ?x)  
  
;;; Query results:  
(last-pair (1 2 3) (3))  
  
;;; Query input:  
(last-pair (2 ?x) (3))  
  
;;; Query results:  
(last-pair (2 3) (3))  
  
;;; Query input:  
(last-pair ?x (3))  
  
;;; Query results:  
(last-pair (3) (3))  
(last-pair (?u-20 3) (3))  
(last-pair (?u-20 ?u-22 3) (3))
```

```
(last-pair (?u-20 ?u-22 ?u-24 3) (3))
(last-pair (?u-20 ?u-22 ?u-24 ?u-26 3) (3))
(last-pair (?u-20 ?u-22 ?u-24 ?u-26 ?u-28 3) (3))
(last-pair (?u-20 ?u-22 ?u-24 ?u-26 ?u-28 ?u-30 3) (3))
(last-pair (?u-20 ?u-22 ?u-24 ?u-26 ?u-28 ?u-30 ?u-32 3) (3))
```

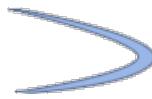
This is using the implementation from Section 4.4.4. (I won't pretend to know exactly what's going on but it seems that rule parts are evaluated in the reverse order to how they are defined, and with the original order the iterative part of the rule is repeatedly evaluated first.)

woofy

Amazing discovery! In fact the two rules are both evaluated alternatively and the second rule has to be evaluated first in order to generate this *infinite stream* result.



# sicp-ex-4.63



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

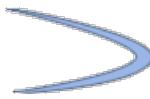
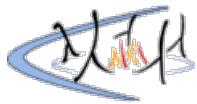
<< Previous exercise (4.62) | Index | Next exercise (4.64) >>

meteorgan

```
rules:  
(assert! (rule (father ?s ?f)  
               (or (son ?f ?s)  
                    (and (son ?w ?s)  
                          (wife ?f ?w)))))  
  
(assert! (rule (grandson ?g ?s)  
               (and (father ?s ?f)  
                     (father ?f ?g))))  
  
;; Query input:  
(grandson Cain ?s)  
  
;; Query output:  
(grandson Cain Irad)  
;; Query input:  
(father ?s Lamech)  
  
;; Query output:  
(father Jubal Lamech)  
(father Jabal Lamech)  
;; Query input:  
(grandson Methusael ?s)  
  
;; Query output:  
(grandson Methusael Jubal)  
(grandson Methusael Jabal)
```

woofy

```
(rule (step-son ?m ?s)  
      (and (son ?w ?s)  
            (wife ?m ?w)))  
  
(rule (has-son ?x ?y)  
      (or (son ?x ?y)  
           (step-son ?x ?y)))  
  
(rule (grandson ?g ?s)  
      (and (has-son ?g ?f)  
            (has-son ?f ?s)))  
  
; the grandson of Cain  
(grandson Cain ?x)  
  
; the sons of Lamech  
(has-son Lamech ?x)  
  
; the grandsons of Methusael  
(grandson Methusael ?x)
```



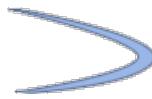
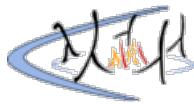
---

[\*\*<< Previous exercise \(4.63\)\*\*](#) | [\*\*Index\*\*](#) | [\*\*Next exercise \(4.65\) >>\*\*](#)

---

meteorgan

first, query (outranked-by (Bitdiddle Ben) ?who), after unifying the conclusion of rule.  
we will evaluate (outranked-by ?middle-manager ?boss), this will query (outranked-by ?  
staff-person ?boss) again, so it will be in infinite loop.



<< Previous exercise (4.64) | Index | Next exercise (4.66) >>

meteorgan

That's because there are four middle-manager whose manager is Warbucks Oliver.

verdammelt

To add more detail:

Because Warbucks supervises:

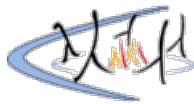
1. Scrooge who supervises Cratchet
2. Bitdiddle who supervises Hacker
3. Bitdiddle who supervises Fect
4. Bitdiddle who supervises Tweakit

Each path is taken and is reported as a separate case.

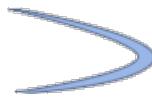
revc

find all unique tuples (variable assignments of the body of rule), but just extract some element(s) from tuple, which leads to the duplicated responses.

```
; ;modified rule
(rule (wheel ?person ?middle-manager ?x)
      (and (supervisor ?middle-manager ?person)
            (supervisor ?x ?middle-manager)))
; ;output
(wheel (Warbucks Oliver) (Scrooge Eben) (Cratchet Robert))
(wheel (Warbucks Oliver) (Bitdiddle Ben) (Tweakit Lem E))
(wheel (Bitdiddle Ben) (Hacker Alyssa P) (Reasoner Louis))
(wheel (Warbucks Oliver) (Bitdiddle Ben) (Fect Cy D))
(wheel (Warbucks Oliver) (Bitdiddle Ben) (Hacker Alyssa P))
```



# sicp-ex-4.66



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.65) | Index | Next exercise (4.67) >>

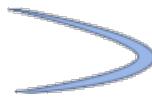
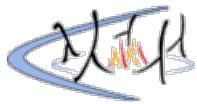
meteorgan

If Ben want to get to sum of salary of wheels, he can you this query:  
`(and (wheel ?who) (salary ?who ?amount))`  
and gets:  
`(and (wheel (Warbucks Oliver)) (salary (Warbucks Oliver) 150000))`  
`(and (wheel (Warbucks Oliver)) (salary (Warbucks Oliver) 150000))`  
`(and (wheel (Bitdiddle Ben)) (salary (Bitdiddle Ben) 60000))`  
`(and (wheel (Warbucks Oliver)) (salary (Warbucks Oliver) 150000))`  
`(and (wheel (Warbucks Oliver)) (salary (Warbucks Oliver) 150000))`  
you can see that Warbucks Oliver's salary occurs four times, so in the sum, Warbucks  
Oliver's salary will be duplicated. Ben can use an unique function to filter the  
duplication in the amount.

woofy

the filtering subject should be keyed on the staff name rather than the amount itself.

Last modified : 2020-05-09 01:40:04  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



---

<< Previous exercise (4.66) | Index | Next exercise (4.68) >>

woofy

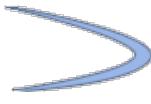
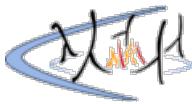
information should at least include : 1. rule name 2. variable bindings, both bounded (to values or other variables) and unbounded

if the rule is already in processing and the 1 and 2 information above is the same with the current attempt to apply after unification, then stop.

user-unknown

<https://github.com/l0stman/sicp/blob/master/4.67.tex>

# sicp-ex-4.68



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.67) | Index | Next exercise (4.69) >>

meteorgan

```
rule:  
(assert! (rule (reverse () ()))  
(assert! (rule (reverse ?x ?y)  
              (and (append-to-form (?first) ?rest ?x)  
                    (append-to-form ?rev-rest (?first) ?y)  
                    (reverse ?rest ?rev-rest))))  
  
(reverse (1 2 3) ?x) : infinite loop  
;;: Query input:  
(reverse ?x (1 2 3))  
  
;;: Query output:  
(reverse (3 2 1) (1 2 3))
```

poly

My solution based on the following procedure.

```
(define (reverse seq)  
  (if (null? seq)  
      ()  
      (append (reverse (cdr seq)) (list (car seq)))))
```

So, there will be two rules:

--1st: an empty list will be got if reverse an empty list.

--2nd: for x y v z, we will get a reversed seq 'z' of (cons x y) only if 'v' is a reversed seq of 'y' and 'z' is (append v '(x))

```
(assert! (rule (reverse () ()))  
  
(assert! (rule (reverse (?x . ?y) ?z)  
                  (and (reverse ?y ?v)  
                        (append-to-form ?v (?x) ?z))))
```

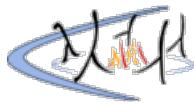
the above one will work for (reverse (1 2 3) ?x), but end up with an infinite loop for (reverse ?x (1 2 3)). But this situation will be opposite if change the order of sequence like:

```
(assert! (rule (reverse () ()))  
  
(assert! (rule (reverse (?x . ?y) ?z)  
                  (and (append-to-form ?v (?x) ?z) ;; changed  
                        (reverse ?y ?v)))) ;;
```

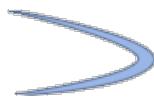
msyb

```
(assert! (reverse () ()) ; It isn't a RULE!!!  
(assert! (rule (reverse (?h . ?t) ?l)  
              (and (reverse ?t ?z)  
                    (append-to-form ?z (?h) ?l))))  
  
;;: Query input:  
(reverse (1 2 3 4) ?x)  
  
;;: Query results:  
(reverse (1 2 3 4) (4 3 2 1))  
  
;;: Query input:  
(reverse ?x (1 2 3 4))  
  
;;: Query results:
```

```
(reverse (4 3 2 1) (1 2 3 4))  
; then infinite loop
```



# sicp-ex-4.69



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.68) | Index | Next exercise (4.70) >>

woofy

```
; Other same stuff from ex_4.64
; ...

(rule (end-with-gs (grandson)))
(rule (end-with-gs (?x . ?y)) (end-wth-gs ?y))
(rule ((grandson) ?x ?y) (grandson ?x ?y))

; Note that the end-with-gs predicate has to be the last one
(rule ((great . ?rel) ?x ?y)
      (and (has-son ?f ?y)
           (?rel ?x ?f)
           (end-with-gs ?rel)))

; Tests:

;;; Query input:
((great grandson) ?g ?ggs)

;;; Query results:
((great grandson) mehujael jubal)
((great grandson) mehujael jabal)
((great grandson) irad lamech)
((great grandson) enoch methushael)
((great grandson) cain mehujael)
((great grandson) adam irad)

;;; Query input:
(?relationship Adam Irad)

;;; Query results:
((great grandson) adam irad)

;;; Query input:
(?relationship Adam Jabal)

;;; Query results:
((great great great great grandson) adam jabal)
```

meteorgan

```
(rule (end-in-grandson (grandson)))
(rule (end-in-grandson (?x . ?rest))
      (end-in-grandson ?rest))

(rule ((grandson) ?x ?y)
      (grandson ?x ?y))
(rule ((great . ?rel) ?x ?y)
      (and (end-in-grandson ?rel)
           (son ?x ?z)
           (?rel ?z ?y)))
```

codybartfast

Rules:  
=====

```
(rule (son-of ?parent ?son)
      (or (son ?parent ?son)
```

```

(and (son ?mother ?son)
      (wife ?parent ?mother)))))

(rule (grandson ?grand-parent ?grand-son)
      (and (son-of ?parent ?grand-son)
            (son-of ?grand-parent ?parent)))))

(rule (same ?x ?x))

(rule ((great . ?rel) ?g ?ggs)
      (or
        (and (same ?rel (grandson))
              (grandson ?i ?ggs)
              (son-of ?g ?i))
        (and (same ?rel (great . ?rest))
              ((great . ?rest) ?i ?ggs)
              (son-of ?g ?i)))))

Output:
=====

;;; Query input:
((great great grandson) Adam ?who)

;;; Query results:
((great great grandson) Adam Mehujael)

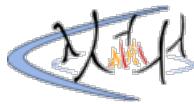
;;; Query input:
((great grandson) ?g ?ggs)

;;; Query results:
((great grandson) Mehujael Jubal)
((great grandson) Mehujael Jabal)
((great grandson) Irad Lamech)
((great grandson) Enoch Methushael)
((great grandson) Cain Mehujael)
((great grandson) Adam Irad)

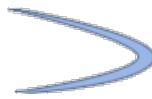
;;; Query input:
(?rel Adam Jabal)

;;; Query results:
((great great great great great grandson) Adam Jabal)

```



# sicp-ex-4.70



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

---

<< Previous exercise (4.69) | Index | Next exercise (4.71) >>

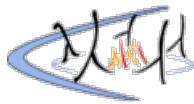
---

meteorgan

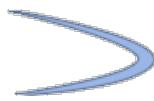
Because we use (cons-stream assertion THE-ASSERTION), so THE-ASSERTIONS will **not** be evaluated, (**set!** THE-ASSERTION (cons-stream assertion THE-ASSERTIONS)) will make THE-ASSERTION in the stream point to itself. so **if** we use THE-ASSERTIONS, it will lead to infinite loop.

---

Last modified : 2012-08-07 02:28:22  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-4.71



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.70) | Index | Next exercise (4.72) >>

meteorgan

```
This will postpone some infinite loop. for example:  
(assert! (married Minnie Mickey))  
(assert! (rule (married ?x ?y)  
                 (married ?y ?x)))  
(married Mickey ?who)  
if we don't use delay, there is no answer to display. but if we use it, we can get:  
;;; Query output:  
(married Mickey Minnie)  
(married Mickey Minnie)  
(married Mickey Minnie)  
....  
this is better than nothing. the reason of this difference is that in this example (apply-  
rules query-pattern frame) will lead to infinite loop, if we delay it, we still can get  
some meaningful answers.
```

ericwen229

As a supplement to the solution of meteorgan, the lazy evaluation here can actually  
avoid infinite loops in some situations. Specifically, when using filters, we don't  
actually use the actual result of a query, since we only care about whether the  
result's empty or not. Consider the following example:

```
(rule (son ?x ?y)  
      (son ?x ?y))
```

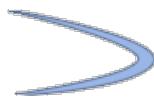
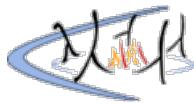
We accidentally define a rule that shares the same name with the assertions. The query (son ?x ?y) is supposed to find all the assertions before get stuck in an infinite loop. But the query (not (son ?x ?y)) can avoid the infinite loop thanks to lazy evaluation.

The example above illustrates a very extreme situation, where a query takes an infinite amount of time. Even if queries do terminate, lazy evaluation can still save a lot of computation. Consider the following example:

```
(not (or <subquery#0>  
        <subquery#1>  
        <subquery#2>  
        ...  
        <subquery#n>))
```

Any subquery with non-empty result can save the effort of processing rest of the subqueries.

<https://github.com/ericwen229/SICP-Solutions>



---

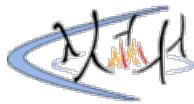
<< Previous exercise (4.71) | Index | Next exercise (4.73) >>

meteorgan

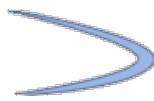
The reason is same to why we use interleave in 3.5.3, it's convenient to **display** infinite stream.

SophiaG

Interleaving is like merging in traffic. You take one from one lane **and** one from the other in order to form a zipper. Otherwise you risk a person sitting at the front of one lane never turning **and** being honked at for all of time. That is one of your infinite lists **if** you attempt to use **append** instead of interleave. For some reason, Californians never figured out merging in traffic either...



# sicp-ex-4.73



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.72) | Index | Next exercise (4.74) >>

meteorgan

This is a same question to exercise 4.71.

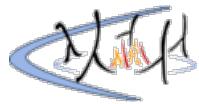
poly

it will end up with an infinite loop when the stream is infinite and we won't get any results. The two arguments (stream-car stream) and (flatten-stream (stream-cdr stream)) will be evaluated first, before the interpreter apply interleave to them. So the flatten-stream will try to get all the elements in the stream, and will never stop until the stream is null.

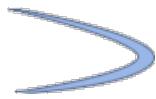
With a finite stream, it will function well, but it has enumerated all the elements in the stream, which means the usage of stream will be meaningless.

```
(define (flatten-stream stream)
  (if (stream-null? stream)
      the-empty-stream
      (interleave
        (stream-car stream)
        (flatten-stream (stream-cdr stream)))))
```

Last modified : 2017-08-29 08:04:38  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-4.74



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

---

[<< Previous exercise \(4.73\)](#) | [Index](#) | [Next exercise \(4.75\) >>](#)

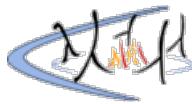
meteorgan

(a)  
`define (simple-flatten stream)
 (stream-map stream-car
 (stream-filter (lambda (s) (not (stream-null? s))) stream)))`

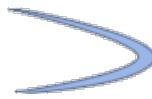
(b)  
no. the order of stream will **not** change the result.

---

Last modified : 2012-08-07 04:03:22  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-4.75



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.74) | Index | Next exercise (4.76) >>

meteorgan

```
; ; add those code
(define (uniquely-asserted pattern frame-stream)
  (stream-flatmap
    (lambda (frame)
      (let ((stream (qevel (negated-query pattern)
                           (singleton-stream frame))))
        (if (singleton-stream? stream)
            stream
            the-empty-stream)))
    frame-stream))
(put 'unique 'qevel uniquely-asserted)

(define (singleton-stream? s)
  (and (not (stream-null? s))
       (stream-null? (stream-cdr s)))))

;;; Query input:
(and (supervisor ?person ?boss) (unique (supervisor ?other ?boss)))

;;; Query output:
(and (supervisor (Cratchet Robert) (Scrooge Eben)) (unique (supervisor (Cratchet Robert)
(Scrooge Eben))))
 (and (supervisor (Reasoner Louis) (Hacker Alyssa P)) (unique (supervisor (Reasoner Louis)
(Hacker Alyssa P))))
```

poly

I don't know whether there is a mistake in the text book. It asks for finding the people with only one superior. But meteorgan just found those with only one subordinate.

I figure that the supervisor of A's supervisor is also A's supervisor. So my solution is as follows.

```
(assert! (rule (staff-with-one-supervisor ?p1 ?p2)
               (and (or (supervisor ?p1 ?p2)
                         (and (supervisor ?p1 ?p3)
                               (supervisor ?p3 ?p2)))
                     (unique (all-supervisors ?p1 ?p))))))

;;; Query input:
(staff-with-one-supervisor ?x ?y)

;;; Quary results:
(staff-with-one-supervisor (aull dewitt) (warbucks oliver))
(staff-with-one-supervisor (scrooge eben) (warbucks oliver))
(staff-with-one-supervisor (bitdiddle ben) (warbucks oliver))
```

msyb

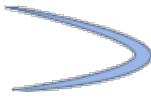
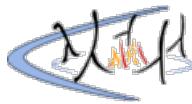
Use the `outranked-by` defined before exercise 4.57.

```
;;; Query input:
(and (supervisor ?person ?boss) (unique (outranked-by ?person ?bosses)))

;;; Query results:
(and (supervisor (Aull DeWitt) (Warbucks Oliver)) (unique (outranked-by (Aull DeWitt)
(Warbucks Oliver))))
 (and (supervisor (Scrooge Eben) (Warbucks Oliver)) (unique (outranked-by (Scrooge Eben)
(Warbucks Oliver))))
 (and (supervisor (Bitdiddle Ben) (Warbucks Oliver)) (unique (outranked-by (Bitdiddle Ben)
(Warbucks Oliver)))))
```



# sicp-ex-4.76



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.75) | Index | Next exercise (4.77) >>

donald

```
(define (new-conjoin conjuncts frame-stream)
  (if (empty-conjunction? conjuncts)
      frame-stream
      (merge (qevel (first-conjunct conjuncts)
                    frame-stream)
             (new-conjoin (rest-conjuncts conjuncts)
                          frame-stream)))))

(define (merge s1 s2)
  (cond ((stream-null? s1) s2)
        ((stream-null? s2) s1)
        (else
         (stream-flatmap (lambda (frame1)
                           (stream-flatmap (lambda (frame2)
                                             (merge-frame frame1 frame2))
                                         s2))
                         s1)))))

(define (merge-frame f1 f2)
  (if (null? f1)
      (singleton-stream f2)
      (let ((b1 (car f1)))
        (let ((b2 (assoc (car b1) f2)))
          (if b2
              (if (equal? (cdr b1) (cdr b2))
                  (merge-frame (cdr f1) f2)
                  the-empty-stream)
              (merge-frame (cdr f1) (cons b1 f2)))))))
```

poly

Actually my solution is not complete. There are still some bugs to be improved.

If we just consider the each clause of 'and' are disjunct, and just separate the evaluation of each clause of 'and' just like the above one solution provided by donald (simply compare the cdr part of two different bindings with same variable in two frames), some input will not be seen as 'legal'. Take an instance:

```
(and (append-to-form (1 2) (3 4) ?x)
      (append-to-form (1) ?y ?x))
```

The implementation of append-to-form is the same as textbook's.

the value of '?y' is based on '?x'. But the '?x' in the second clause is apparently not bound. Its value should be seen as the same as '?x' in the first clause. But we just separate the two clauses. So it won't work and the input above will get nothing. The detail is kind of complicated.

The evaluation of first clause will get:

```
((? 12 z) 3 4) ((? 14 y) 3 4) ((? 11 z) (? 12 u) ? 12 z) ((? 12 y) 3 4) ((? 12 v)) ((? 12 u) . 2) ((? x) (? 11 u) ? 11 z) ((? 11 y) 3 4) ((? 11 v) 2) ((? 11 u) . 1))
```

second is:

```
((? 15 z) ? 17 y) ((? 15 y) ? 17 y) ((? x) (? 15 u) ? 15 z) ((? y) ? 15 y) ((? 15 v)) ((? 15 u) . 1))
```

As we can see, the '?x' binding in first one is ((? x) (? 11 u) ? 11 z), which is ((? x) (? 15 u) ? 15 z) in the second one. Apparently it will not be seen as the same if we simply compare its cdr part.

Actually, in some complex case, the value of bindings in the frame will always also be variables as above situation. So I think we need to find its final actual binding and extend it to the frame if there is no not-variable bindings.

the procedure "extend-if-possible" provided in the textbook will do this job:

```
(define (extend-if-possible var val frame)
  (let ((binding (binding-in-frame var frame)))
    (cond (binding
```

```

(unify-match
  (binding-value binding) val frame))
; var has no binding check if val is variable.
((var? val)
  (let ((binding (binding-in-frame val frame)))
    (if binding
        ; check if var and binding are matched
        (unify-match
          var (binding-value binding) frame)
        ; bind var to val if both of them are variable
        (extend var val frame))))
; check if var itself is in val
((depends-on? val var frame)
  'failed)
(else (extend var val frame))))))

```

My whole solution is as follows.

```

(define (conjoin conjuncts frame-stream)
  (if (empty-conjunction? conjuncts)
      frame-stream
      (merge
        (qevel (first-conjunct conjuncts) frame-stream)
        (conjoin (rest-conjuncts conjuncts) frame-stream)))))

(define (merge stream1 stream2)
  (stream-flatmap
    (lambda (f1)
      (stream-filter
        (lambda (f) (not (eq? f 'failed)))
        (stream-map
          (lambda (f2) (merge-frames f1 f2))
          stream2)))
    stream1))

(define (merge-frames f1 f2)
  (cond ((null? f1) f2)
        ((eq? 'failed f2) 'failed)
        (else
          (let ((var (binding-variable (car f1)))
                (val (binding-value (car f1))))
            (let ((extension (extend-if-possible var val f2)))
              (merge-frames (cdr f1) extension))))))

;; Query input:
(and (append-to-form (1 2) (3 4) ?x)
      (append-to-form (1) ?y ?x))

;; Quary results:
(and (append-to-form (1 2) (3 4) (1 2 3 4))
      (append-to-form (1) (2 3 4) (1 2 3 4)))

```

Actually, when there are some kind of recursions existed, this will end up with an infinite loop. Take an instance:

```

(rule (reverse () ()))
(rule (reverse (?x . ?y) ?z)
  (and (reverse ?y ?v)
    (append-to-form ?v (?x) ?z)))

```

The above one is just one bug. Another one is for the consideration of 'not' and 'unique' procedure we define before. As we know this two is for filtering the result we find out. But there will be no frame to filter if we separate the evaluation of each clauses of 'and'.

That's all.

nopnopnoo

my version, with original "not" [expamle: (and query1 (not query2))],its not working well.Because "not" is a filter,it only can reduce a frame-stream, and cant increase a a frame-stream.

ps:<unify-pattern == unify-match>

```

(define (extend-if-possible_frame binding data-frame)
  (let ((opposition (binding-in-frame (binding-variable binding) data-frame)))
    (if opposition
        (unify-pattern (binding-value binding) (binding-value opposition) data-
frame)
        (extend-with-binding binding data-frame)))
(define (unify-frame cast-frame data-frame)

```

```

(cond ((eq? data-frame 'failed) 'failed)
      ((null? cast-frame) data-frame)
      (else (unify-frame (cdr cast-frame)
                           (extend-if-possible-frame (car cast-frame) data-
frame))))
(define (and-frame-stream frame-stream1 frame-stream2)
  (if (or (null-stream? frame-stream1) (null-stream? frame-stream2))
      null-stream
      (interleave-stream-delayed (flatmap-stream (lambda (frame)
                                                    (let ((unified-frame
(unify-frame (car-stream frame-stream1)
frame)))
(if (eq? unified-
frame 'failed)
null-stream
(singleton-
stream unified-frame)))))))
(define (conjoin conjuncts frame-stream)
  ;; 平行的计算所有conjunct的frame-stream并合并它们,但这有个问题,即像not,lisp-value这些"过滤器"会可能会导致匹配失败,因为当输入为空framestream时,输出也为空frame-stream.
  (define (conjoin-frame frame)
    (define (loop conjuncts conjuncted-frame-stream)
      (if (null? conjuncts)
          conjuncted-frame-stream
          (loop (rest-conjuncts conjuncts)
                (and-frame-stream conjuncted-frame-stream
                                  (qeval (first-conjunct conjuncts)
                                         (singleton-stream frame))))))
    (loop conjuncts (singleton-stream '())))
  (flatmap-stream conjoin-frame frame-stream))
  (put! 'and 'qeval conjoin)

```

closeparen

One of the most challenging exercises in the book for me! I conceptualize it as: first evaluate all the conjuncts against the given frame-stream. Then produce the cartesian product of all the resulting frames. Then prune the cartesian product to only those frames that can be reconciled with each other.

```

;; stream-cartesian takes a regular list of streams and produces
;; (a1 b1) (a1 b2) (a1 b3) (a2 b1) (a2 b2) ....
(define (stream-cartesian streams)
  (define (prepend x)
    (stream-map (lambda (y) (cons x y)) (stream-cartesian (cdr streams)))))

  (if (null? streams)
      (singleton-stream the-empty-stream)
      (stream-flatmap prepend (car streams)))))

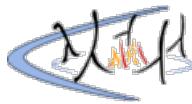
;; reconcile-frames takes a list of frames and reconciles them against the
;; "with" bindings
(define (reconcile-frames frames with)
  (define (reconcile bindings with)
    (if (null? bindings)
        with
        (let* ((first-binding (car bindings))
               (next (extend-if-consistent (binding-variable first-binding)
                                           (binding-value first-binding)
                                           with)))
          (if (eq? next 'failed)
              '()
              (reconcile (cdr bindings) next)))))

    (reconcile (fold-left append '() frames) with))

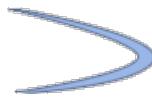
  (define (conjoin-parallel conjuncts frame-stream)
    (let* ((evaluated-streams (map (lambda (conjunct) (qeval conjunct frame-stream))
                                    conjuncts))
           (cartesian (stream-cartesian evaluated-streams))
           (solve-frame (lambda (frame-from-stream)
                         (stream-map (lambda (frames-from-cartesian)
                                       (reconcile-frames frames-from-cartesian
                                         frame-from-stream))
                                     cartesian)))
           (solved (stream-flatmap solve-frame frame-stream)))))


```

```
(stream-filter (lambda (x) (not (null? x))) solved)))
```



# sicp-ex-4.77



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.76) | Index | Next exercise (4.78) >>

poly

```
(define (filter? exp)
  (or (list-value? exp)
      (not? exp)))

(define (conjoin conjuncts frame-stream)
  (conjoin-mix conjuncts '() frame-stream))

(define (conjoin-mix conjs delayed-conjs frame-stream)
  (if (null? conjs)
      (if (null? delayed-conjs)
          frame-stream ; conjoin finish if both of conjuncts are empty
          the-empty-stream) ; no result return cause filters with unbound vars exist
      (let ((first (first-conjunct conjs))
            (rest (rest-conjuncts conjs)))
        (if (filter? first)
            (let ((check-result
                  (conjoin-check first delayed-conjs frame-stream)))
              (conjoin-mix rest
                           (car check-result)
                           (cdr check-result)))
            (let ((new-frame-stream (qevel first frame-stream)))
              (let ((delayed-result
                    (conjoin-delayed delayed-conjs '() new-frame-stream)))
                (conjoin-mix rest (car delayed-result) (cdr delayed-result)))))))

(define (conjoin-delayed delayed-conjs rest-conjs frame-stream)
  ; evaluate those conjuncts in delayed-conjs if there are
  ; enough bindings for them.
  (if (null? delayed-conjs)
      (cons rest-conjs frame-stream)
      (let ((check-result
            (conjoin-check (first-conjunct delayed-conjs)
                          rest-conjs frame-stream)))
        (conjoin-delayed (cdr delayed-conjs)
                        (car check-result)
                        (cdr check-result)))))

(define (conjoin-check target conjs frame-stream)
  ; Check if there are any unbound vars in target.
  ; Delay it if there are unbound vars, or just evaluate it.
  (if (has-unbound-var? (contents target) (stream-car frame-stream))
      (cons (cons target conjs) frame-stream)
      (cons conjs (qevel target frame-stream)))))

(define (has-unbound-var? exp frame)
  (define (tree-walk exp)
    (cond ((var? exp)
           (let ((binding (binding-in-frame exp frame)))
             (if binding
                 (tree-walk (binding-value binding))
                 true)))
           ((pair? exp)
            (or (tree-walk (car exp)) (tree-walk (cdr exp))))
            (else false)))
    (tree-walk exp))
```

unique is also sort of filter, but won't work with this because the contents of unique usually contain at least one variable that is not relevant with other variables.

revc

No complicated mechanism is required.

we can simply rearrange the order of clauses of compound queries by putting all filters at the end, which is an efficient and trivial method. In order to accomplish this, we will normalize the non-normalized compound queries during the parse phase of qeval.

```
;;; Exercise 4.77

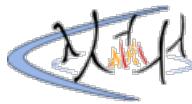
(define compound-table '())
(define (put-compound combinator) (set! compound-table (cons combinator compound-table)))
(define (compound? query) (memq (type query) compound-table))

(define filter-table '())
(define (put-filter operator) (set! filter-table (cons operator filter-table)))
(define (filter? query) (memq (type query) filter-table))

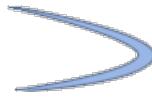
(define (normalize clauses)
  (let ((filters (filter filter? clauses))
        (non-filters (filter (lambda (x) (not (filter? x))) clauses)))
    (append non-filters filters)))

(define (qeval query frame-stream)
  (let ((qproc (get (type query) 'qeval)))
    (cond ((compound? query) (qproc (normalize (contents query)) frame-stream))
          (qproc (qproc (contents query) frame-stream))
          (else (simple-query query frame-stream)))))

(put-compound 'and)
(put-filter 'not)
(put-filter 'lisp-value)
(put-filter 'unique)
```



# sicp-ex-4.78



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (4.77) | Index | Next exercise (4.79) >>

poly

<https://github.com/collins562/SICP-solutions/tree/master/Chapter%204-Metalinguistic%20Abstraction/4.Logical%20Programming/4.78>

```
revc      ;;; Exercise 4.78
          (load "ch4-ambeval.scm")

(define input-prompt ";;; Query input:")
(define output-prompt ";;; Query results:")

(define (qevel query frame succeed fail)
  ((analyze query) frame succeed fail))

;;; predicates
(define (and? query) (tagged-list? query 'and))
(define (or? query) (tagged-list? query 'or))
(define (not? query) (tagged-list? query 'not))
(define (lisp-value? query) (tagged-list? query 'lisp-value))

(define (analyze query)
  (cond ((and? query) (analyze-and (contents query)))
        ((or? query) (analyze-or (contents query)))
        ((lisp-value? query) (analyze-lisp-value (contents query)))
        ((not? query) (analyze-not (contents query)))
        (else (analyze-simple query)))))

(define (analyze-lisp-value call)
  (lambda (frame succeed fail)
    (if (execute
          (instantiate
            call
            frame
            (lambda (v f)
              (error "Unknown pat var -- LISP-VALUE" v))))
        (succeed frame fail)
        (fail)))))

(define (analyze-not operands)
  (lambda (frame succeed fail)
    ((analyze (negated-query operands))
     frame
     (lambda (ext fail2)
       (fail)))
     (lambda () (succeed frame fail)))))

(define (analyze-or disjuncts)
  (lambda (frame succeed fail)
    (define (try)
      ((analyze (car disjuncts))
       frame
       succeed
       (lambda ()
         ((analyze-or (cdr disjuncts))
          frame succeed fail)))))

    (if (empty-disjunction? disjuncts)
        (succeed frame fail)
        (try)))))

(define (analyze-and conjuncts)
  (lambda (frame succeed fail)
    (define (try)
      ((analyze (car conjuncts))
       frame
       (lambda (ext fail2)
         ((analyze-and (cdr conjuncts))
          ext succeed fail2))
       fail))))
```

```

(if (empty-conjunction? conjuncts)
    (succeed frame fail)
    (try)))

;;;; rewritten
(define (rule-body rule)
  (if (null? (cddr rule))
      #f
      (caddr rule)))

(define (analyze-simple query)
  (lambda (frame succeed fail)
    (define (try-assertion assertions)
      (if (stream-null? assertions)
          (try-rule (fetch-rules query frame))
          (let ((ext (pattern-match query (stream-car assertions) frame)))
            (fail2 (lambda () (try-assertion (stream-cdr assertions))))))
        (if (succeeded? ext)
            (succeed ext fail2)
            (fail2)))))

(define (try-rule rules)
  (if (stream-null? rules)
      (fail)
      (let* ((clean-rule (rename-variables-in (stream-car rules)))
             (ext (unify-match query (conclusion clean-rule) frame))
             (fail2 (lambda () (try-rule (stream-cdr rules)))))
        (if (succeeded? ext)
            (if (rule-body clean-rule)
                (qeval (rule-body clean-rule)
                      ext
                      succeed fail2)
                (succeed ext fail2))
            (fail2)))))

  (try-assertion (fetch-assertions query frame))
))

(define (driver-loop)
  (define (internal-loop try-again)
    (prompt-for-input input-prompt)
    (let ((q (query-syntax-process (read))))
      (cond
        ;;; ((eq? q '#!eof) 'goodbye!)
        ((eq? q 'try-again) (try-again))
        ((assertion-to-be-added? q)
         (add-rule-or-assertion! (add-assertion-body q))
         (newline)
         (display "Assertion added to assertions base."))
        (else (newline)
              (display ";;;; Starting a new problem ")
              (qeval q
                     '() ; an empty frame
                     ;; ambeval success
                     (lambda (val next-alternative)
                       (announce-output output-prompt)
                       (user-print
                         (instantiate q
                                       val
                                       (lambda (v f)
                                         (contract-question-mark v))))))
                     (internal-loop next-alternative)))
        ;;; ambeval failure
        (lambda ()
          (announce-output
            ";;;; There are no more values of")
          (user-print q)
          (driver-loop))))))

  (internal-loop
    (lambda ()
      (newline)
      (display ";;;; There is no current problem")
      (driver-loop)))))

;;;;;;
; test
;;;;;;

;;;; Query input:
(big-shot ?p ?q)

;;;; Starting a new problem
;;;; Query results:
(big-shot (Aull DeWitt) administration)

```

```

;;; Query input:
try-again

;;; Query results:
(big-shot (Cratchet Robert) accounting)

;;; Query input:
try-again

;;; Query results:
(big-shot (Scrooge Eben) accounting)

;;; Query input:
try-again

;;; Query results:
(big-shot (Scrooge Eben) accounting)

;;; Query input:
(reverse (1 2 3) ?x)

;;; Starting a new problem
;;; Query results:
(reverse (1 2 3) (3 2 1))

;;; Query input:
(and (replace ?p2 ?p1)
      (salary ?p1 ?s1)
      (salary ?p2 ?s2)
      (lisp-value > ?s2 ?s1))

;;; Starting a new problem
;;; Query results:
(and (replace (Hacker Alyssa P) (Fect Cy D)) (salary (Fect Cy D) 35000) (salary (Hacker
Alyssa P) 40000) (lisp-value > 40000 35000))

;;; Query input:
try-again

;;; Query results:
(and (replace (Warbucks Oliver) (Aull DeWitt)) (salary (Aull DeWitt) 25000) (salary
(Warbucks Oliver) 150000) (lisp-value > 150000 25000))

;;; Query input:
try-again

;;; There are no more values of
(and (replace (? p2) (? p1)) (salary (? p1) (? s1)) (salary (? p2) (? s2)) (lisp-value > (? s2) (? s1)))

```

**Q:** You will probably also find, however, that your new query language has subtle differences in behavior from the one implemented here. Can you find examples that illustrate this difference?

**A:** See ``(big-shot ?p ?q)`` (There are many duplicate answers).

woofy

Here I think the author is asking us to implement the query language interpreter as a user APPLICATION program on top of the *underlying host* non deterministic (amb) language rather than as an EXTENSION to it as poly and revc did above.

For example, some key modification to highlight with:

```

; either match an assertion or a query
(define (simple-query query-pattern frame)
  (amb (find-assertions query-pattern frame)
       (apply-rules query-pattern frame)))

; match any of the OR disjuncts
(define (disjoin disjuncts frame)
  (qeval (an-element-of disjuncts) frame))

; match any of the assertions
(define (find-assertions pattern frame)
  (let ((datum (an-element-of (fetch-assertions pattern frame))))
    (check-an-assertion datum pattern frame)))

; apply any of the rules

```

```

(define (apply-rules pattern frame)
  (let ((rule (an-element-of (fetch-rules pattern frame))))
    (apply-a-rule rule pattern frame))

; if-fail from ex_4.52.
; The idea is to ensure failure or else we trackback.
; It might be better to implement a special form require-fail in the amb evaluator for
this which I also implemented in the code link below
(define (negate operands frame)
  (let ((result 'failed))
    (if-fail
      (begin
        (qeval (negated-query operands) frame)
        (permanent-set! result 'success) ; at least one match found
        (amb)) ; exhaust the alternatives of success matches
      (if (eq? result 'failed)
          frame
          (amb)))))

; use require to filter lisp-value
(define (lisp-value call frame)
  (require
    (execute
      (instantiate
        call
        frame
        (lambda (v f)
          (error "Unknown pat var -- LISP-VALUE" v))))))

frame)

```

That's basically it. Note that we use if-fail construct from exercise 4.52 to deal with NOT clause. Another place we need to pay attention to is that we should call (amb) directly instead of tag 'failed frames during pattern matching or unification.

Another interesting thing to notice is that we can have two driver loops at play during runtime, with the logic driver loop being nested in the Amb driver loop! See my [complete code](#) for walking through the solution.

The difference in this none deterministic paradigm with the original stream approach is that (using the amb evaluator the book provided) we cannot deal with infinite outputs or output the answers in an interleaving style as in the stream approach.

# sicp-ex-4.79

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.78) | Index | Next exercise (5.1) >>

SophiaG

Switching from a quick-and-dirty variable renaming function to an environment structure like the rest of Scheme would have to involve an implementation that solves the problem of carrying around frames that are never evaluated. I would propose what Daniel P. Friedman and David S. Wise referred to as "Suicidal Suspensions" in their seminal 1976 paper on non-strict evaluation in Lisp, "Cons Should Not Evaluate Its Arguments," where they proved such a method would not involve more calls to eval/apply than McCarthy's strict interpreter. This would entail terminating all unevaluated environments once a given query has returned, which means both that the system should use the occasion to batch evaluate this set of environments (ideally using the linear-time tree-based search from exercise 4.76), as well as checking for repetition in its history in order to prevent infinite loops (along the lines of exercise 4.67), as either would be particularly disastrous once the offending environments have been disposed of.

lockywolf

I think this exercise can be shamelessly ripped off the Indiana Technical Report on eu-Prolog: <https://legacy.cs.indiana.edu/ftp/techreports/TR155.pdf>

Haha, nope, I lied, they also rename variables.

As a food for thought, there is "Logic Programming: A Classified Bibliography", also I found something similar in "On Implementing Prolog In Functional Programming" by Mats Carlsson.

There is also an attempted solution by skanev:

<https://github.com/skanev/playground/blob/master/scheme/sicp/04/79.scm> , but I think that he misses the point, although I am not sure.

Last modified : 2020-03-21 08:49:03  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

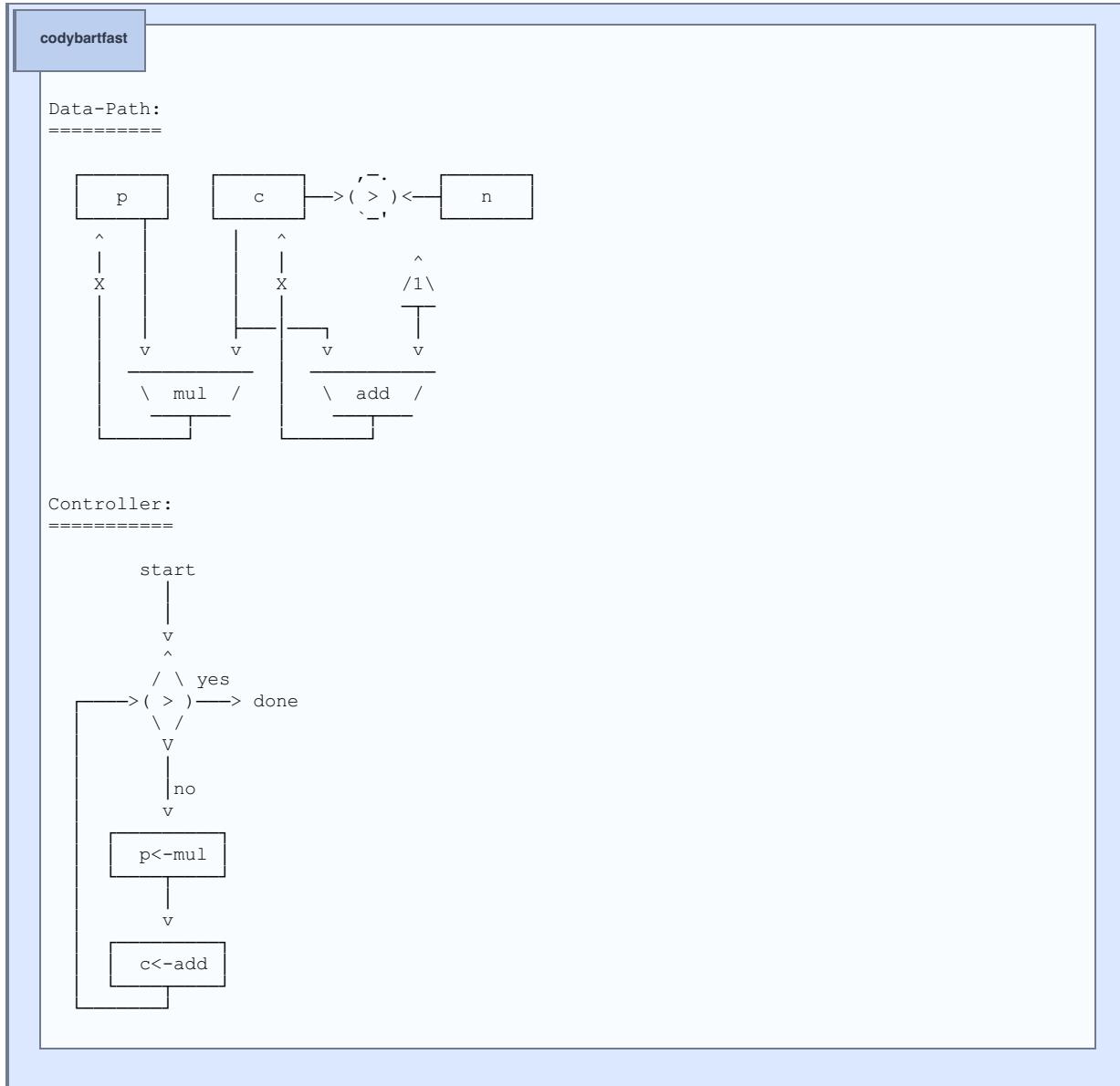
# sicp-ex-5.1

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (4.79) | Index | Next exercise (5.2) >>



# sicp-ex-5.2

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (5.1) | Index | Next exercise (5.3) >>

meteorgan

```
(define fact-machine
  (make-machine
    '(c p n)
    '(list (list '* *) (list '+ +) (list '> >))
    '((assign c (const 1))
      (assign p (const 1)))
    test-n
    (test (op >) (reg c) (reg n))
    (branch (label fact-done))
    (assign p (op *) (reg c) (reg p))
    (assign c (op +) (reg c) (const 1))
    (goto (label test-n))
    fact-done))

(set-register-contents! fact-machine 'n 5)
(start fact-machine)

(get-register-contents fact-machine 'p)
=>120
```

verdammelt

```
(controller
  (assign p (const 1))
  (assign c (const 1))

  test-n
  (test (op >) (reg c) (reg n))
  (branch (label done))
  (assign x (op *) (reg p) (reg c))
  (assign y (op +) (reg c) (const 1))
  (assign p (reg x))
  (assign c (reg y))
  (goto (label test-n))

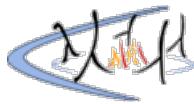
  done)
```

asd

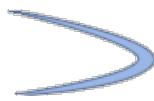
```
(controller
  (assign Prod (const 1))
  (assign Count (const 1))

  test-n
  (test (op >) (reg Count) (reg n))
  (branch (label done))
  (assign t (op *) (reg Prod) (reg Count))
  (assign Prod (reg t))
  (assign t (op +) (reg Count) (const 1))
  (assign Prod (reg t))
  (goto (label test-n))
  done)
```





# sicp-ex-5.3



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.2) | Index | Next exercise (5.4) >>

meteorgan

```
(define sqrt-machine
  (make-machine
    '(x guess temp)
    '(list (list '---) (list '< <) (list '/ /) (list '+ +) (list '* *) (list '> >))
    '((assign guess (const 1.0))
      test-g
      (assign temp (op *)) (reg guess) (reg guess))
     (assign temp (op -) (reg temp) (reg x))
     (test (op >) (reg temp) (const 0))
     (branch (label iter))
     (assign temp (op -) (const 0) (reg temp)))
    iter
    (test (op <) (reg temp) (const 0.001))
    (branch (label sqrt-done))
    (assign temp (op /) (reg x) (reg guess))
    (assign temp (op +) (reg temp) (reg guess))
    (assign guess (op /) (reg temp) (const 2))
    (goto (label test-g))
    sqrt-done)))
  (set-register-contents! sqrt-machine 'x 2)
  (start sqrt-machine)

  (get-register-contents sqrt-machine 'guess)
=>1.4142156862745097
```

verdammelt

```
; ; 1. assuming that good-enough? and improve are primitives

(controller
  (assign x (op read))
  (assign guess (const 1.0))

  test-good
  (test (op good-enough?) (reg guess) (reg x))
  (branch (label done))
  (assign t (op improve) (reg guess) (reg x))
  (assign guess (reg t))
  (goto test-good)
  done

  (perform (op print) (reg guess))
  )

; ; 2. inline improve

(controller
  (assign x (op read))
  (assign guess (const 1.0))

  test-good
  (test (op good-enough?) (reg guess) (reg x))
  (branch (label done))

; ; improve procedure
  (assign div (op /) (reg x) (reg guess))
  (assign avg (op average) (reg guess) (reg div))

  (assign t (reg avg))
  (assign guess (reg t))
  (goto test-good)
  done

  (perform (op print) (reg guess))
  )

; ; 2a inline average
```

```

(controller
  (assign x (op read))
  (assign guess (const 1.0))

  test-good
  (test (op good-enough?) (reg guess) (reg x))
  (branch (label done))

  ; improve procedure
  (assign div (op /) (reg x) (reg guess))

  ; average procedure
  (assign sum (op +) (reg guess) (reg div))
  (assign avg (op /) (reg sum) (const 2))

  (assign t (reg avg))
  (assign guess (reg t))
  (goto test-good)
done

(perform (op print) (reg guess))
)

3. inline good-enough?
(controller
  (assign x (op read))
  (assign guess (const 1.0))

  test-good

  ; good-enough? procedure
  (assign square (op *) (reg guess) (reg guess))
  (assign diff (op -) (reg square) (reg x))
  (test (op <) (reg diff) (const 0))
  (branch test-abs-neg)
  test-abs-pos
  (assign abs (reg diff))
  test-abs-neg
  (assign abs (op *) (reg diff) (const -1))
  (test (op <) (reg abs) (const 0.001))

  (branch (label done))

  ; improve procedure
  (assign div (op /) (reg x) (reg guess))
  ; average procedure
  (assign sum (op +) (reg guess) (reg div))
  (assign avg (op /) (reg sum) (const 2))

  (assign t (reg avg))
  (assign guess (reg t))
  (goto test-good)
done

(perform (op print) (reg guess))
)

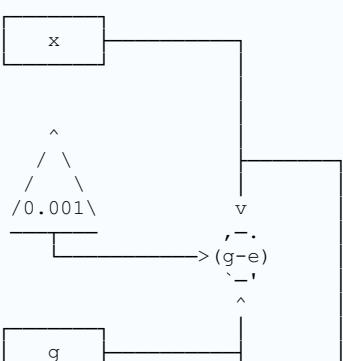
```

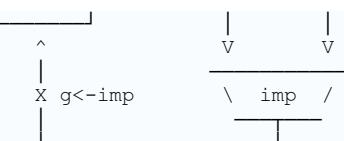
codybartfast

With good-enough? And improve  
=====

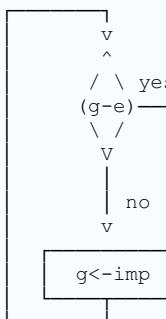
Data-path:

-----





Controller Diagram:

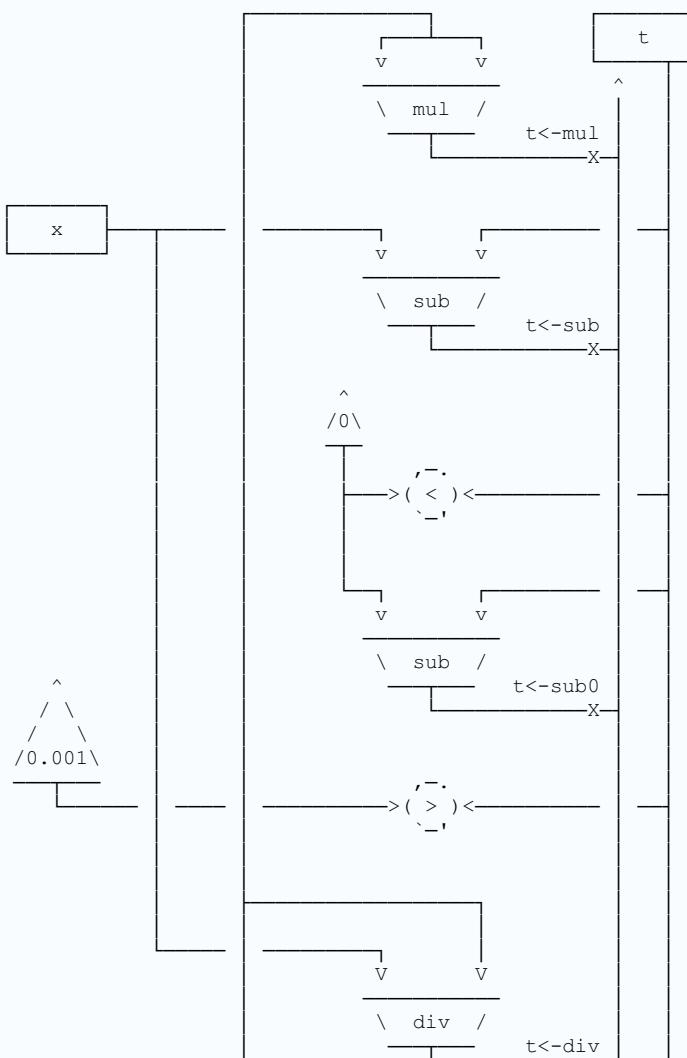


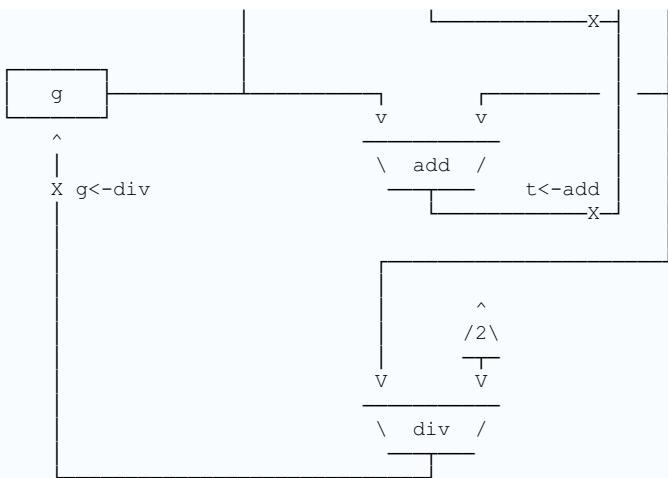
Controller:

```
(controller
  test-g-e
    (test (op g-e) (reg g) (reg x) (const 0.001))
    (branch (label sqrt-done))
    (assign g (op imp) (reg g) (reg x))
    (goto (label test-g-e))
  sqrt-done)
```

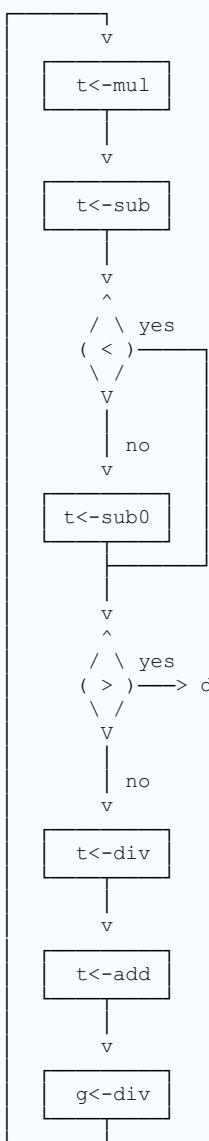
#### Arithmetic Only

Data-path:





Controller Diagram:



Controller:

```
(controller
  test-g-e
    (assign t (op mul) (reg g) (reg g))
    (assign t (op sub) (reg x) (reg t))
    (test (op <) (const 0) (reg t))
    (branch (label test-g-e-final))
    (assign t (op sub) (const 0) (reg t))
  test-g-e-final
    (test (op >) (const 0.001) (reg t))
    (branch (label sqrt-done))
    (assign t (op div) (reg x) (reg g))
```

```
(assign t (op add) (reg g) (reg t))
(assign g (op div) (reg t) (const 2))
(goto (label test-g-e))
sqrt-done)
```

anon

Why does no one use abs procedure in their description? I don't think that it is correct to omit this step

# sicp-ex-5.4

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (5.3) | Index | Next exercise (5.5) >>

meteorgan

```
(a)
(define expt-machine
  (make-machine
    '(b n r continue)
    (list (list '=) (list '* *) (list '- -))
    '(((assign continue (label expt-done))
       expt-loop
       (test (op =) (reg n) (const 0))
       (branch (label expt-base))
       (save continue)
       (assign n (op -) (reg n) (const 1))
       (assign continue (label after-expt))
       (goto (label expt-loop)))
      after-expt
      (restore continue)
      (assign r (op *) (reg r) (reg b))
      (goto (reg continue))
      expt-base
      (assign r (const 1))
      (goto (reg continue))
      expt-done)))
  (set-register-contents! expt-machine 'b 2)
  (set-register-contents! expt-machine 'n 10)
  (start expt-machine)

  (get-register-contents expt-machine 'r)
=> 1024

(b)
(define expt-machine
  (make-machine
    '(product b n product)
    (assign product (const 1))
    expt-loop
    (test (op =) (reg n) (const 0))
    (branch (label expt-done))
    (assign product (op *) (reg product) (reg b))
    (assign n (op -) (reg n) (const 1))
    (goto (label expt-loop))
    (goto (expt-loop))
    expt-done))
```

codybartfast

Recursive Exponentiation

=====

Controller:

-----

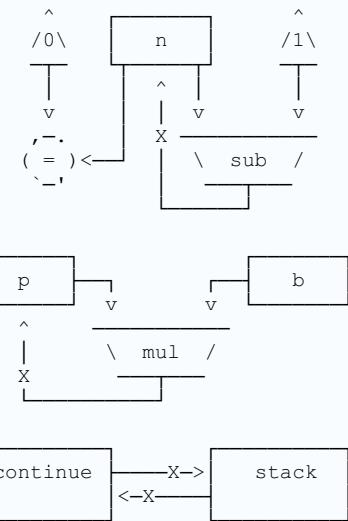
```
(controller
  (assign continue (label expn-done))
  test
  (test (op =) (reg n) (const 0))
  (branch (label base-case))
  (save continue)
  (assign continue (label after-expn))
  (assign n (op sub) (reg n) (const 1))
  (goto (label test)))
  after-expn
  (restore continue))
```

```

(assign p (op mul) (reg p) (reg b))
(goto (reg continue))
base-case
  (assign p (const 1))
  (goto (reg continue))
expn-done)

```

Data-path:



Iterative Exponentiation

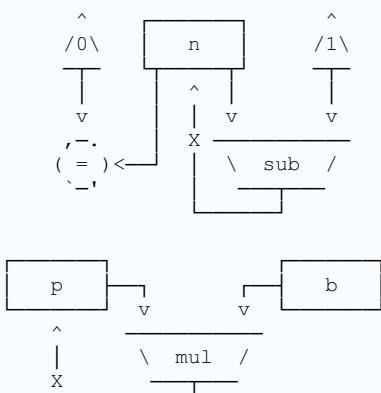
Controller:

```

(controller
  test
    (test (op =) (reg n) (const 0))
    (branch (label expn-done))
    (assign p (op mul) (reg p) (reg b))
    (assign n (op sub) (reg n) (const 1))
    (goto (label test))
  expn-done)

```

Data-path:



revc

[https://drive.google.com/file/d/1EtHjkm2wZaJ8LCu\\_NULYSYh55XXtRTTQ/view?usp=sharing](https://drive.google.com/file/d/1EtHjkm2wZaJ8LCu_NULYSYh55XXtRTTQ/view?usp=sharing)

anon

This recursive controller uses only 3 registers.

```

;;;;;;
(controller
  (assign continue (label expt-done))
  (assign b (op read))
  (assign n (op read)))

```

```
expt-loop
(test (op =) (const 0) (reg n))
(branch (label base-case))
(save continue)
(assign continue (label after-expt))
(assign n (op -) (reg n) (const 1))
(goto (label expt-loop))
after-expt
	restore continue)
(assign n (op *) (reg b) (reg n))
(goto (reg continue))
base-case
(assign n (const 1))
(goto (reg continue))
expt-done
(perform (op print) (reg n)))

(controller
(assign b (op read))
(assign counter (op read))
(assign product (const 1))
expt-loop
(test (op =) (reg counter) (const 0))
(branch (label expt-done))
(assign counter (op -) (reg counter) (const 1))
(assign product (op *) (reg b) (reg product))
(goto (label expt-loop))
expt-done
(perform (op print) (reg product)))
```

# sicp-ex-5.5

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (5.4) | Index | Next exercise (5.6) >>

meteorgan

```
using the example of factorial in the book.  
(set-register-contents! machine 'n 2)  
  
1. n = 2  
(save continue) ;; stack: ((label fact-done))  
(save n) ;; stack (2 (label fact-done))  
(assign n (op -) (reg n) (const 1)) ;; n =1  
(assign continue (label after-fact)) ;; continue: (label after-fact)  
  
2. n =1. go to base-case  
(restore n) ;; n: 2; stack: ((label fact-done))  
(restore continue) ;; continue: label fact-done stack: ()
```

codybartfast

Factorial

=====

State for factorial 4 at each label:

fact-loop	fact-loop	fact-loop	fact-loop
n=4 c=done	n=3 c=aft	n=2 c=aft	n=1 c=aft
-----	-----	-----	-----
-----	fact-done <	fact-done <	fact-done <
-----	4 <	4 <	4 <
-----	-----	after-fact <	after-fact <
-----	-----	3 <	3 <
-----	-----	-----	after-fact <
-----	-----	-----	2 <
-----	-----	-----	-----

base-case	after-fact	after-fact	after-fact
n=1 c=aft	n=1 c=aft v=1	n=2 c=aft v=2	n=3 c=aft v=6
-----	-----	-----	-----
fact-done	fact-done	fact-done	fact-done
4	4	4	4
after-fact	after-fact	after-fact	-----
3	3	3	-----
after-fact	after-fact	-----	-----
2	2	-----	-----
-----	-----	-----	-----

fact-done	fact-done	fact-done	fact-done
n=4 c=done v=24	-----	-----	-----
-----	-----	-----	-----

Fibonacci

=====

State for 4th Fibonacci at each label

-----	-----	-----	-----
-------	-------	-------	-------

fib-loop	fib-loop	fib-loop	fib-loop
-----	-----	-----	-----
n=4 c=done	n=3 c=afn-1	n=2 c=afn-1	n=1 c=afn-1
-----	-----	-----	-----
-----	fib-done < 4	fib-done < 4	fib-done < 4
-----	-----	afterfib-n-1 < 3	afterfib-n-1 < 3
-----	-----	-----	afterfib-n-1 < 2
-----	-----	-----	-----
-----	-----	-----	-----
immediate-answer	afterfib-n-1	fib-loop	immediate-answer
-----	-----	-----	-----
n=1 c=afn-1	n=1 c=afn-1 v=1	n=0 c=afn-2 v=1	n=0 c=afn-2 v=1
-----	-----	-----	-----
fib-done 4	fib-done 4	fib-done 4	fib-done 4
afterfib-n-1 3	afterfib-n-1 3	afterfib-n-1 3	afterfib-n-1 3
afterfib-n-1 2	afterfib-n-1 2	afterfib-n-1 1	afterfib-n-1 1
-----	-----	-----	-----
-----	-----	-----	-----
afterfib-n-2	afterfib-n-1	fib-loop	immediate-answer
-----	-----	-----	-----
n=0 c=afn-2 v=0	n=0 c=afn-1 v=1	n=1 c=afn-2 v=1	n=1 c=afn-2 v=1
-----	-----	-----	-----
fib-done 4	fib-done 4	fib-done 4	fib-done 4
afterfib-n-1 3	afterfib-n-1 3	afterfib-n-1 1	afterfib-n-1 1
afterfib-n-1 1	-----	-----	-----
-----	-----	-----	-----
-----	-----	-----	-----
afterfib-n-2	afterfib-n-1	fib-loop	fib-loop
-----	-----	-----	-----
n=1 c=afn-2 v=1	n=1 c=afn-1 v=2	n=2 c=afn-2 v=2	n=1 c=afn-1 v=2
-----	-----	-----	-----
fib-done 4	fib-done 4	fib-done < 2	fib-done < 2
afterfib-n-1 1	-----	-----	afterfib-n-2 < 2
-----	-----	-----	-----
-----	-----	-----	-----
immediate-answer	afterfib-n-1	fib-loop	immediate-answer
-----	-----	-----	-----
n=1 c=afn-1 v=2	n=1 c=afn-1 v=1	n=0 c=afn-2 v=1	n=0 c=afn-2 v=1
-----	-----	-----	-----
fib-done 2	fib-done 2	fib-done 2	fib-done 2
afterfib-n-2 2	afterfib-n-2 2	afterfib-n-2 1	afterfib-n-2 1
-----	-----	-----	-----
-----	-----	-----	-----
afterfib-n-2	afterfib-n-2	fib-done	-----
-----	-----	-----	-----
n=0 c=afn-2 v=0	n=0 c=afn-2 v=1	n=1 c=done v=3	-----
-----	-----	-----	-----
fib-done 2	fib-done 2	-----	-----
afterfib-n-2 1	-----	-----	-----
-----	-----	-----	-----

Newly pushed values marked with '<'  
Wish I had picked 3 instead of 4.

# sicp-ex-5.6

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

<< Previous exercise (5.5) | Index | Next exercise (5.7) >>

---

meteorgan

in afterfib-n-1, (restore continue) (save continue). those two instructions just pop the  
continue **and** then place it in the stack.

---

Last modified : 2012-08-09 13:51:06  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-5.7

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

[<< Previous exercise \(5.6\)](#) | [Index](#) | [Next exercise \(5.8\) >>](#)

---

meteorgan

```
this is another test for expt-machine:  
(set-register-contents! expt-machine 'b 3)  
(set-register-contents! expt-machine 'n 4)  
(start expt-machine)  
'done  
'done  
'done  
> (get-register-contents expt-machine 'r)  
81
```

---

Last modified : 2012-08-09 13:56:53  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**

# sicp-ex-5.8

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (5.7) | Index | Next exercise (5.9) >>

meteorgan

1.

the value in register a is 3.

```
(define (label-exist? labels label-name)
  (assoc label-name labels))

(define (extract-labels text)
  (if (null? text)
      '()
      (cons '() '())
      (let ((result (extract-labels (cdr text))))
        (let ((insts (car result)) (labels (cdr result)))
          (let ((next-inst (car text)))
            (if (symbol? next-inst)
                (if (label-exist? labels next-inst)
                    (error "the label has existed EXTRACT-LABELS" next-labels)
                    (cons insts
                          (cons (make-label-entry next-inst insts) labels)))
                (cons (cons (make-instruction next-inst) insts)
                      labels)))))))
```

donald

1. the value of a is 3

```
(define (extract-labels text receive)
  (if (null? text)
      (receive '() '())
      (extract-labels (cdr text)
                     (lambda (insts labels)
                       (let ((next-inst (car text)))
                         (if (symbol? next-inst)
                             (let ((s (assoc next-inst labels)))
                               (if s
                                   (error "Repeated label name" next-inst)
                                   (receive insts
                                         (cons (make-label-entry next-inst
                                                               insts)
                                               labels))))
                             (receive (cons (make-instruction next-inst)
                                           insts)
                                   labels)))))))
```

# sicp-ex-5.9

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (5.8) | Index | Next exercise (5.10) >>

lockywolf

```
(compile-lambda ...)
```

from section 5.5.2 subsection "compiling lambda expressions" doesn't work with this exercise.

meteorgan

```
; ; add a test to make-operation-exp
(define (make-operation-exp expr machine labels operations)
  (let ((op (lookup-prim (operation-exp-op expr) operations)))
    (aprocs
      (map (lambda (e)
              (if (label-exp? e)
                  (error "can't operate on label -- MAKE-OPERATION-EXP" e)
                  (make-primitive-exp e machine labels)))
           (operation-exp-operands expr))))
    (lambda ()
      (apply op (map (lambda (p) (p)) aprocs))))
```

codybartfast

```
(define (make-operation-exp exp machine labels operations)
  (let ((op (lookup-prim (operation-exp-op exp) operations)))
    (aprocs
      (map (lambda (e)
              (if (or (register-exp? e) (constant-exp? e))
                  (make-primitive-exp e machine labels)
                  (error "Invalid Argument for operation -- ASSEMBLE" e)))
           (operation-exp-operands exp))))
    (lambda ()
      (apply op (map (lambda (p) (p)) aprocs))))
```

# sicp-ex-5.10

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (5.9) | Index | Next exercise (5.11) >>

meteorgan

```
; ; add an inc instruction. (inc <register-name>) , add 1 to the value of register-name.

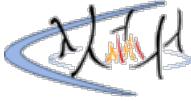
(define (make-execution-procedure inst labels machine
                                     pc flag stack ops)
  (cond ((eq? (car inst) 'assign)
         (make-assign inst machine labels ops pc))
        ((eq? (car inst) 'test)
         (make-test inst machine labels ops flag pc))
        ((eq? (car inst) 'branch)
         (make-branch inst machine labels flag pc))
        ((eq? (car inst) 'goto)
         (make-goto inst machine labels pc))
        ((eq? (car inst) 'save)
         (make-save inst machine stack pc))
        ((eq? (car inst) 'restore)
         (make-restore inst machine stack pc))
        ((eq? (car inst) 'inc)
         (make-inc inst machine pc))
        ((eq? (car inst) 'perform)
         (make-perform inst machine labels ops pc))
        (else (error "Unknown instruction type -- ASSEMBLE"
                     inst)))))

(define (make-inc inst machine pc)
  (let ((target
         (get-register machine (inc-reg-name inst))))
    (lambda ()
      (set-contents! target (+ (get-contents target) 1))
      (advance-pc pc)))
  (define (inc-reg-name inst)
    (cadr inst)))
```

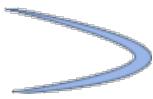
verdammelt

```
; ; as an example - instead of (assign x (op +) (reg a) (reg b)) I want
; ; the syntax: (assign x (apply + (reg a) (reg b)))

(define (operation-exp? exp)
  (tagged-list? exp 'apply))
(define (operation-exp-op exp)
  (cadr exp))
(define (operation-exp-operands exp)
  (cddr exp))
```



# sicp-ex-5.11



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

[<< Previous exercise \(5.10\) | Index | Next exercise \(5.12\) >>](#)

meteorgan

```

(a)
(assign n (reg val))
	restore val)
we can use (restore n) replace these two instructions. Because val contain Fib(n-2),
(restore n) make n containing Fib(n-1), then (assign val (op +) (reg val) (reg n)) works
still.

(b)
;; in make-save, push the register name and contents on the stack.
(define (make-save inst machine stack pc)
  (let ((reg (get-register machine
                           (stack-inst-reg-name inst))))
    (lambda ()
      (push stack (cons (stack-inst-reg-name inst) (get-contents reg)))
      (advance-pc pc)))))

;; when pop stack, check if the register name on the stack is equal to
;; reg-name in (restore reg-name)
(define (make-restore inst machine stack pc)
  (let* ((reg-name (stack-inst-reg-name inst))
         (reg (get-register machine reg-name)))
    (lambda ()
      (let ((pop-reg (pop stack)))
        (if (eq? (car pop-reg) reg-name)
            (begin
              (set-contents! reg (cdr pop-reg))
              (advance-pc pc))
            (error "the value is not from register:" reg-name))))))

(c)
;; first modify make-stack making it including the name of register.
;; all the named-register are in s, we can use assoc to find it.
;; add interface add-reg-stack to add register to stack.
(define (make-stack)
  (let ((s '()))
    (define (push reg-name value)
      (let ((reg (assoc reg-name s)))
        (if reg
            (set-cdr! reg (cons value (cdr reg)))
            (error "the register is not in the stack -- PUSH" reg-name))))
    (define (pop reg-name)
      (let ((reg (assoc reg-name s)))
        (if reg
            (if (null? (cdr reg))
                (error "Empty stack for register -- POP" reg-name)
                (let ((top (cadr reg)))
                  (set-cdr! reg (cddr reg))
                  top))
            (error "the register is not in the stack -- POP" reg-name))))
    (define (add-reg-stack reg-name)
      (if (assoc reg-name s)
          (error "this register is already in the stack -- ADD-REG-STACK" reg-name)
          (set! s (cons (list reg-name) s))))
    (define (initialize)
      (for-each
        (lambda (stack)
          (set-cdr! stack '()))
        s)
      'done)
    (define (dispatch message)
      (cond ((eq? message 'push) push)
            ((eq? message 'pop) pop)
            ((eq? message 'add-reg-stack) add-reg-stack)
            ((eq? message 'initialize) (initialize))
            (else (error "Unknown request -- STACK" message))))
    dispatch))

(define (add-reg-stack stack reg-name)

```

```
((stack 'add-reg-stack) reg-name))

;; change make-store and make-restore, because their parameters had change.
(define (make-save inst machine stack pc)
  (let ((reg (get-register machine
                           (stack-inst-reg-name inst))))
    (lambda ()
      (push stack (stack-inst-reg-name inst) (get-contents reg))
      (advance-pc pc)))))

(define (make-restore inst machine stack pc)
  (let* ((reg-name (stack-inst-reg-name inst))
         (reg (get-register machine reg-name)))
    (lambda ()
      (let ((value (pop stack reg-name)))
        (set-contents! reg value)
        (advance-pc pc)))))

;; modify allocate-register in make-new-machine, when allocate a register,
;; add it to stack.
(define (allocate-register name)
  (if (assoc name register-table)
      (error "Multiply defined register: " name)
      (begin
        (add-reg-stack stack name)
        (set! register-table
              (cons (list name (make-register name))
                    register-table)))
      'register-allocated))
```

**verdammelt**

;; I chose a different path than shown above. I removed the machine's stack and instead made every register have a stack.

```
(define (make-register name)
  (let ((contents '*unassigned*)
        (stack (make-stack)))
    (define (dispatch message)
      (cond ((eq? message 'get) contents)
            ((eq? message 'set)
             (lambda (value) (set! contents value)))
            ((eq? message 'push)
             ((stack 'push) contents))
            ((eq? message 'restore)
             (set! contents (stack 'pop)))
            (else
             (error "Unknown request -- REGISTER" message))))
      dispatch))

(define (make-save inst machine stack pc)
  (let ((reg (get-register machine
                           (stack-inst-reg-name inst))))
    (lambda ()
      (reg 'push)
      (advance-pc pc)))))

(define (make-restore inst machine stack pc)
  (let ((reg (get-register machine
                           (stack-inst-reg-name inst))))
    (lambda ()
      (reg 'restore)
      (advance-pc pc)))))

;; 1. remove the stack parameter being passed around everywhere
;; 2. don't create a stack in the machine
;; 3. snippet of make-new-machine that needs to change
(let ((register-table
       (list (list 'pc pc) (list 'flag flag))))
  (let ((the-ops
         (list (list 'initialize-stack
                     (lambda ()
                       (map (lambda (s) (stack 'initialize))
                            register-table)))))))
    ...)))
```

**aos**

I took a different path than everyone else... I ended up creating a `stack-table` similar to the register table.

`; Allocate a stack`

```
(define (allocate-stack name)
```

```

(if (assoc name stack-table)
  (error "Multiply defined stacks: " name)
  (set! stack-table
    (cons
      (list name
            (make-stack))
      stack-table)))
  'stack-allocated)

;; Then when initializing machine in (make-machine ...)

(for-each (lambda (reg-name) ; New stack for each register
              ((machine 'allocate-stack)
               reg-name))
          register-names)

;; Initializing all stacks becomes this:
(the-ops
  (list (list 'initialize-stack
              (lambda ()
                (for-each (lambda (stack)
                            (stack 'initialize))
                  (map cadr stack-table)))))))

```

revc	;; c. alt.1
------	-------------

```

(define (make-save inst machine stack pc)
  (let ((reg (get-register machine
                           (stack-inst-reg-name inst))))
    (lambda ()
      (push stack (cons reg (get-contents reg))) ;**
      (advance-pc pc)))))

(define (pop stack reg)
  (let loop ((items '())
            (top (stack 'pop)))
    (if (eq? (car top) reg)
        (begin (for-each (lambda (e) (push stack e)) (reverse items)) top)
        (loop (cons top items) (stack 'pop)))))

(define (make-restore inst machine stack pc)
  (let ((reg (get-register machine
                           (stack-inst-reg-name inst))))
    (lambda ()
      (set-contents! reg (pop stack reg)) ;**
      (advance-pc pc)))))

;;; c. alt.2

(define (make-machine register-names ops controller-text)
  (let ((machine (make-new-machine)))
    (for-each (lambda (register-name)
                (((machine 'allocate-register) register-name)
                 ((machine 'allocate-stack) register-name))) ;++
                register-names)
    ((machine 'install-operations) ops)
    ((machine 'install-instruction-sequence)
     (assemble controller-text machine))
    machine))

(define (make-new-machine)
  (let ((pc (make-register 'pc))
        (flag (make-register 'flag))
        ;; (stack (make-stack))
        (the-instruction-sequence '()))
    (let ((the-ops
           '())
         (register-table
           (list (list 'pc pc) (list 'flag flag)))
         (stack-table
           '())))
      (define (allocate-register name)
        (if (assoc name register-table)
            (error "Multiply defined register: " name)
            (set! register-table
              (cons (list name (make-register name))
                    register-table)))
        'register-allocated)

      (define (allocate-stack name) ;++
        (if (assoc name stack-table)
            (error "Multiply defined register: " name)
            (set! stack-table

```

```

        (cons (list name (make-stack))
              stack-table)))
      'stack-allocated)

(define (lookup-register name)
  (let ((val (assoc name register-table)))
    (if val
        (cadr val)
        (error "Unknown register:" name)))

(define (lookup-stack name) ;**
  (let ((val (assoc name stack-table)))
    (if val
        (cadr val)
        (error "Unknown stack:" name)))

(define (execute)
  (let ((insts (get-contents pc)))
    (if (null? insts)
        'done
        (begin
          ((instruction-execution-proc (car insts)))
          (execute)))))

(define (dispatch message)
  (cond ((eq? message 'start)
         (set-contents! pc the-instruction-sequence)
         (execute))
        ((eq? message 'install-instruction-sequence)
         (lambda (seq) (set! the-instruction-sequence seq)))
        ((eq? message 'allocate-register) allocate-register)
        ((eq? message 'allocate-stack) allocate-stack);**
        ((eq? message 'get-register) lookup-register)
        ((eq? message 'get-stack) lookup-stack);**
        ((eq? message 'install-operations)
         (lambda (ops) (set! the-ops (append the-ops ops))))
        ((eq? message 'stack) stack)
        ((eq? message 'operations) the-ops)
        (else (error "Unknown request -- MACHINE" message)))))

(dispatch))

(define (update-insts! insts labels machine)
  (let ((pc (get-register machine 'pc))
        (flag (get-register machine 'flag))
        ;; (stack (machine 'stack))
        (ops (machine 'operations)))
    (for-each
     (lambda (inst)
       (set-instruction-execution-proc!
        inst
        (make-execution-procedure
         (instruction-text inst) labels machine
         pc flag ops));*
       insts)))
  insts))

(define (make-execution-procedure inst labels machine
                                   pc flag ops)
  (cond ((eq? (car inst) 'assign)
         (make-assign inst machine labels ops pc))
        ((eq? (car inst) 'test)
         (make-test inst machine labels ops flag pc))
        ((eq? (car inst) 'branch)
         (make-branch inst machine labels flag pc))
        ((eq? (car inst) 'goto)
         (make-goto inst machine labels pc))
        ((eq? (car inst) 'save)
         (make-save inst machine pc));**
        ((eq? (car inst) 'restore)
         (make-restore inst machine pc));**
        ((eq? (car inst) 'perform)
         (make-perform inst machine labels ops pc))
        (else (error "Unknown instruction type -- ASSEMBLE"
                     inst)))))

(define (get-stack machine reg-name)
  ((machine 'get-stack) reg-name))

(define (make-save inst machine pc)
  (let* ((reg-name (stack-inst-reg-name inst))
         (reg (get-register machine reg-name))
         (stack (get-stack machine reg-name)))
    (lambda ()
      (push stack (get-contents reg))
      (advance-pc pc)))))

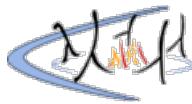
(define (make-restore inst machine stack pc)
  (let* ((reg-name (stack-inst-reg-name inst))
         (reg (get-register machine reg-name)))

```

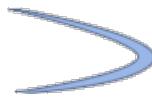
```
(stack (get-stack machine reg-name)))
(lambda ()
  (set-contents! reg (pop stack))
  (advance-pc pc)))
```

---

Last modified : 2020-01-28 08:52:17  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-5.12



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.11) | Index | Next exercise (5.13) >>

ypeels

```
(define (make-dataset)
  (let ((dataset '()))
(define (adjoin! datum)
  (if (not (is-in-dataset? datum))
      (set! dataset (cons datum dataset))))
(define (print)
  (display dataset)
  (newline))
(define (is-in-dataset? datum) ; private helper function
  (cond
    ((symbol? datum) (memq datum dataset))
    ((list? datum) (member datum dataset))
    (else (error "Unknown data type -- IS-IN-dataset?" datum))))
(define (dispatch message)
  (cond
    ((eq? message 'adjoin!) adjoin!)
    ((eq? message 'print) (print))
    (else (error "Unknown operation -- DATASET" message))))
  dispatch))

(define (adjoin-to-dataset! datum dataset)
  ((dataset 'adjoin!) datum))

(define (print-dataset dataset)
  (dataset 'print))

; the rest of this solution "overrides" existing functions in ch5-regsim.scm
; that is, they add a bit of functionality, then call the existing functions.
; this way, readers (the author included) don't have to compare code
; just to figure out what was added/changed.

; implemented as a "facade" in front of the old machine
(define (make-new-machine-5.12)
  (let ((machine-regsim (make-new-machine-regsim)) ; "base object" or "delegate"
        (dataset-table
          (list
            (list 'assign (make-dataset))
            (list 'branch (make-dataset))
            (list 'goto (make-dataset))
            (list 'perform (make-dataset))
            (list 'restore (make-dataset))
            (list 'save (make-dataset))
            (list 'test (make-dataset))

            (list 'goto-registers (make-dataset))
            (list 'save-registers (make-dataset))
            (list 'restore-registers (make-dataset)))))

; register names are determined by the user, so these should be stored separately
; sure, it'd take one sick cookie to name a register 'goto',
; but a register named 'test' is not inconceivable.
; also, a user could technically manipulate pc and flag directly
  (assign-dataset-table
    (list
      (list 'pc (make-dataset))
      (list 'flag (make-dataset)))))

; "public procedures"
(define (allocate-register-5.12 name)
  (set! assign-dataset-table
    (cons ; no duplicate checking - original regsim will crash on that anyway
      (list name (make-dataset))
      assign-dataset-table))
  ((machine-regsim 'allocate-register) name))

(define (lookup-dataset name)
```

```

        (lookup-dataset-in-table name dataset-table))

(define (lookup-assign-dataset name)
  (lookup-dataset-in-table name assign-dataset-table))

(define (print-all-datasets)
  (print-dataset-table dataset-table "Instructions and registers used")
  (print-dataset-table assign-dataset-table "Assignments"))

; "private procedures" (cannot be invoked from outside the object)
(define (lookup-dataset-in-table name table)
  (let ((val (assoc name table)))
    (if val
        (cadr val)
        (error "dataset not found -- GET-DATASET-FROM-TABLE" name table)))))

(define (print-dataset-table table title)
  (newline)
  (display title)
  (newline)
  (for-each
    (lambda (table-entry)
      (display (car table-entry))
      (display ": ")
      (print-dataset (cadr table-entry)))
    table))

; expose public API
(define (dispatch message)
  (cond
    ; one override
    ((eq? message 'allocate-register) allocate-register-5.12)

    ; new messages
    ((eq? message 'print-all-datasets) (print-all-datasets))
    ((eq? message 'lookup-dataset) lookup-dataset)
    ((eq? message 'lookup-assign-dataset) lookup-assign-dataset)

    ; punt everything else to "base class" / delegate - INCLUDING error handling
    (else (machine-regsim message))))
  dispatch)

(define (make-execution-procedure-5.12 inst labels machine pc flag stack ops)
  (let ((dataset ((machine 'lookup-dataset) (car inst)))
        (adjoin-to-dataset! (cdr inst) dataset))
    (make-execution-procedure-regsim inst labels machine pc flag stack ops)))

(define (make-goto-5.12 inst machine labels pc)
  (let ((dest (goto-dest inst))) ; duplicated 2 lines of supporting logic
    (if (register-exp? dest)
        (let ((dataset ((machine 'lookup-dataset) 'goto-registers)))
          (adjoin-to-dataset! (register-exp-reg dest) dataset)))
    (make-goto-regsim inst machine labels pc)) ; punt to ch5-regsim.scm

(define (make-save-5.12 inst machine stack pc)
  (let ((dataset ((machine 'lookup-dataset) 'save-registers)))
    (adjoin-to-dataset! (stack-inst-reg-name inst) dataset))
  (make-save-regsim inst machine stack pc))

(define (make-restore-5.12 inst machine stack pc)
  (let ((dataset ((machine 'lookup-dataset) 'restore-registers)))
    (adjoin-to-dataset! (stack-inst-reg-name inst) dataset))
  (make-restore-regsim inst machine stack pc))

(define (make-assign-5.12 inst machine labels operations pc)
  (let ((dataset ((machine 'lookup-assign-dataset) (assign-reg-name inst))))
    (adjoin-to-dataset! (assign-value-exp inst) dataset))
  (make-assign-regsim inst machine labels operations pc))

; -----
; example usage.

(load "ch5-regsim.scm")

; make the overrides official.
(define make-new-machine-regsim make-new-machine)
(define make-new-machine make-new-machine-5.12)

(define make-goto-regsim make-goto)
(define make-goto make-goto-5.12)

```

```

(define make-save-regsim make-save)
(define make-save make-save-5.12)

(define make-restore-regsim make-restore)
(define make-restore make-restore-5.12)

(define make-assign-regsim make-assign)
(define make-assign make-assign-5.12)

(define make-execution-procedure-regsim make-execution-procedure)
(define make-execution-procedure make-execution-procedure-5.12)

(define fib-machine (make-machine :register-names ops controller-text
  '(n val continue)
  (list (list '< <) (list '- -) (list '+ +))
  '(
    ; from ch5.scm
    (assign continue (label fib-done))
    fib-loop
    (test (op <) (reg n) (const 2))
    (branch (label immediate-answer))
    ;;; set up to compute Fib(n-1)
    (save continue)
    (assign continue (label afterfib-n-1))
    (save n) ; save old value of n
    (assign n (op -) (reg n) (const 1)); clobber n to n-1
    (goto (label fib-loop)) ; perform recursive call
    afterfib-n-1 ; upon return, val contains Fib(n-1)
    (restore n)
    (restore continue)
    ;;; set up to compute Fib(n-2)
    (assign n (op -) (reg n) (const 2))
    (save continue)
    (assign continue (label afterfib-n-2))
    (save val); save Fib(n-1)
    (goto (label fib-loop))
    afterfib-n-2 ; upon return, val contains Fib(n-2)
    (assign n (reg val)); n now contains Fib(n-2)
    (restore val); val now contains Fib(n-1)
    (restore continue)
    (assign val ; Fib(n-1)+Fib(n-2)
      (op +) (reg val) (reg n))
    (goto (reg continue)); return to caller, answer is in val
    immediate-answer
    (assign val (reg n))
    (goto (reg continue))
    fib-done)))
  )

(fib-machine 'print-all-datasets)

```

### Rptx

```

; added ((machine 'gather-info) controller-text) in `make-machine
; added ((eq? message 'gather-info) (lambda (controller-text)
;   (gather-info controller-text)))
; added ((eq? message 'get-info) (lambda (type) (assoc type information)))
; to `make-new-machine
; use:
; ((<machine> 'get-info) <info-type>)
; for example:
; ((recursive-fib-machine 'get-info) 'entry-points)
; ((<machine> 'get-info) <reg-name>))

; to maintain abstraction.
(define (get-info machine info)
  ((machine 'get-info) info))

(define (gather-info controller-text)
  (define (gather inst-type insts)
    (if debug
        (format #t "\n ~a \n" insts))
    (define (gather-iter gathered left lst)
      (if debug
          (format #t "gather-iter: \n"
                  gathered: ~a \n
                  lst: ~a \n" gathered lst))
      (cond ((null? lst) (list (cons inst-type gathered)
                                left))
            ((not (pair? (car lst)))
             (gather-iter gathered left (cdr lst)))
            ((eq? inst-type
                  (caar lst))
             (if (member (car lst)
                        gathered)

```

```

(gather-iter gathered left (cdr lst))
(gather-iter (cons (car lst) gathered)
              left (cdr lst))))
(else
  (gather-iter gathered (cons (car lst) left)
                (cdr lst))))
(gather-iter '() '() insts))
(define (gather-entry-points gotos)
  (define (gather-iter gathered lst)
    (if (null? lst)
        (list 'entry-points gathered)
        (let ((dest (goto-dest (car lst))))
          (if (register-exp? dest)
              (gather-iter (cons (register-exp-reg dest)
                                gathered)
                          (cdr lst))
              (gather-iter gathered (cdr lst))))))
    (list (gather-iter '() gotos)))
(define (gather-saved-reg saved)
  (list (cons 'stacked-ref (map (lambda (x)
                                   (stack-inst-reg-name x)) saved))))
(define (gather-sources assigns)
(define (sources-iter reg gathered left lst)
  (cond ((null? lst)
         (list (list reg gathered) left))
        ((eq? reg (assign-reg-name (car lst)))
         (sources-iter reg
                       (cons (cddr (car lst))
                             gathered)
                       left
                       (cdr lst)))
        (else
         (sources-iter reg gathered
                       (cons (car lst)
                             left)
                       (cdr lst)))))

(define (sources-loop insts)
  (if (null? insts)
      '()
      (let* ((reg (assign-reg-name (car insts)))
             (srcs (sources-iter reg '() '()
                                  insts)))
        (cons (car srcs)
              (sources-loop (cadr srcs))))))
(sources-loop assigns)
(let* ((assigns
        (gather 'assign controller-text))
       (tests
        (gather 'test (cadr assigns)))
       (branches
        (gather 'branch (cadr tests)))
       (gotos
        (gather 'goto (cadr branches)))
       (saves
        (gather 'save (cadr goto)))
       (restores
        (gather 'restore (cadr saves)))
       (performs
        (gather 'perform (cadr restores)))
       (entry-points
        (gather-entry-points (cdar goto)))
       (stacked
        (gather-saved-reg (cdar saves)))
       (sources
        (gather-sources (cdar assigns)))
       (registers
        (list
         (cons 'registers (map car sources))))))
  (append
   (map (lambda (x)
          (car x))
        (list assigns tests branches goto saves
              restores performs)
        entry-points stacked sources registers)))

```

aos

My solution can be found at:  
<https://github.com/aois/SICP/blob/master/exercises/ch5/2/ex-12.ss>

codybartfast

```
Sample results for Fibonacci machine:

Instructions:
=====
((assign val (reg n))
 (assign val (op +) (reg val) (reg n))
 (assign n (reg val))
 (assign continue (label afterfib-n-2))
 (assign n (op -) (reg n) (const 2))
 (assign n (op -) (reg n) (const 1))
 (assign continue (label afterfib-n-1))
 (assign continue (label fib-done))
 (branch (label immediate-answer))
 (goto (reg continue))
 (goto (label fib-loop))
 (restore val)
 (restore continue)
 (restore n)
 (save val)
 (save n)
 (save continue)
 (test (op <) (reg n) (const 2)))

Entry Registers:
=====
(continue)

Stack Registers:
=====
(n continue val)

Register Sources:
=====
((continue ((label fib-done)
            (label afterfib-n-1)
            (label afterfib-n-2)))
 (n (((op -) (reg n) (const 1))
      ((op -) (reg n) (const 2))
      (reg val)))
 (val (((op +) (reg val) (reg n))
       (reg n))))
```

code: <https://github.com/codybartfast/sicp/blob/master/chapter5/machine-12.scm#L485>

# sicp-ex-5.13

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (5.12) | Index | Next exercise (5.14) >>

meteorgan

```
In make-new-machine, change the code of lookup-register
(define (lookup-register name)
  (let ((val (assoc name register-table)))
    (if val
        (cadr val)
        (begin
          (allocate-register name)
          (lookup-register name)))))
```

aos

Damn it... that's so much simpler than what I was doing. I actually ended up scanning out all the register names and allocating them during assembly.

```
; Extracts all the registers in a single instruction
(define (extract-regs instruction regs)
  (cond ((null? instruction) '())
        ((and (pair? (car instruction))
              (eq? (caar instruction) 'reg)
              (not (memq (cadar instruction) regs)))
         (cons (cadar instruction)
               (extract-regs (cdr instruction) regs)))
        (else (extract-regs (cdr instruction) regs)))))

;; While extracting the labels and instructions,
; we also extract the registers
(define (extract-labels text receive)
  (if (null? text)
      (receive '() '())
      (extract-labels
       (cdr text)
       (lambda (insts labels regs)
         (let ((next-inst (car text)))
           (if (symbol? next-inst) ; label
               (receive
                insts
                (cons (make-label-entry next-inst
                                         insts)
                      labels)
                regs)
               (receive
                 (cons (make-instruction next-inst)
                       insts)
                 labels
                 ; We only include registers we have not found yet
                 (append (extract-regs next-inst regs)
                         regs)))))))

(define (assemble controller-text machine)
  (extract-labels controller-text
    (lambda (insts labels regs)
      ;; Allocate all the extract regs during assembly
      (for-each
       (lambda (reg)
         ((machine 'allocate-register) reg))
       regs)
      (update-insts! insts
                    labels
                    machine)
      insts)))
```

# sicp-ex-5.14

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

<< Previous exercise (5.13) | Index | Next exercise (5.15) >>

---

meteorgan

fact-machine first push n **and** continue to the stack, totally  $2(n-1)$  times, in this process, there is no pop operations. then pop stack. so the number-pushes **and** max-depth are both  $2(n-1)$ .

---

Last modified : 2012-08-12 01:00:57  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**

# sicp-ex-5.15

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (5.14) | Index | Next exercise (5.16) >>

meteorgan

```
; change the code in make-new-machine, they had been marked.
(define (make-new-machine)
  (let ((pc (make-register 'pc))
        (flag (make-register 'flag))
        (stack (make-stack))
        (the-instruction-sequence '()))
    (instruction-number 0)) ; ***
(define (print-instruction-number)
  (display (list "current instruction number is: " instruction-number))
  (set! instruction-number 0)
  (newline))
(let ((the-ops
       (list (list 'initialize-stack
                  (lambda () (stack 'initialize)))
             (list 'print-stack-statistics
                   (lambda () (stack 'print-statistics)))))
         (register-table
          (list (list 'pc pc) (list 'flag flag))))
  (define (allocate-register name)
    (if (assoc name register-table)
        (error "Multiply defined register: " name)
        (set! register-table
              (cons (list name (make-register name))
                    register-table)))
  'register-allocated)
(define (lookup-register name)
  (let ((val (assoc name register-table)))
    (if val
        (cadr val)
        (begin
          (allocate-register name)
          (lookup-register name))))
(define (execute)
  (let ((insts (get-contents pc)))
    ((instruction-execution-proc (car insts)))
     (set! instruction-number (+ instruction-number 1)) ; ***
     (execute)))
(define (dispatch message)
  (cond ((eq? message 'start)
         (set-contents! pc the-instruction-sequence)
         (execute))
        ((eq? message 'install-instruction-sequence)
         (lambda (seq) (set! the-instruction-sequence seq)))
        ((eq? message 'allocate-register) allocate-register)
        ((eq? message 'get-register) lookup-register)
        ((eq? message 'install-operations)
         (lambda (ops) (set! the-ops (append the-ops ops))))
        ((eq? message 'instruction-number) print-instruction-number)
        ;;***
        ((eq? message 'stack) stack)
        ((eq? message 'operations) the-ops)
        (else (error "Unknown request -- MACHINE" message)))
  dispatch))
```

revc

There is a roughly linear relationship between the number of instructions executed and input n.

But in fact, as the input n increases, the ratio also gradually increases, which shows that it is not a strictly linear relationship between the number of instructions executed and input n.

```
(define (make-stack)
  (let ((s '())
        (number-pushes 0)
        (max-depth 0)
        (current-depth 0))
```

```

(define (push x)
  (set! s (cons x s))
  (set! number-pushes (+ 1 number-pushes))
  (set! current-depth (+ 1 current-depth))
  (set! max-depth (max current-depth max-depth)))
(define (pop)
  (if (null? s)
      (error "Empty stack -- POP" 'pop)
      (let ((top (car s)))
        (set! s (cdr s))
        (set! current-depth (- current-depth 1))
        top)))
(define (initialize)
  (set! s '())
  (set! number-pushes 0)
  (set! max-depth 0)
  (set! current-depth 0)
  'done)
(define (print-statistics)
  (newline)
  (for-each display (list "total-pushes: " number-pushes
                          "\n"
                          "maximum-depth: " max-depth
                          "\n"
                          )))
(define (dispatch message)
  (cond ((eq? message 'push) push)
        ((eq? message 'pop) (pop))
        ((eq? message 'initialize) (initialize))
        ((eq? message 'print-statistics)
         (print-statistics))
        (else
         (error "Unknown request -- STACK" message))))
  dispatch))

(define (make-new-machine)
  (let ((pc (make-register 'pc))
        (flag (make-register 'flag))
        (stack (make-stack))
        (the-instruction-sequence '()))
    (the-instruction-counter 0))
  (let ((the-ops
         (list (list 'initialize-stack
                     (lambda () (stack 'initialize)))
               ;**next for monitored stack (as in section 5.2.4)
               ;-- comment out if not wanted
               (list 'print-stack-statistics
                     (lambda () (stack 'print-statistics))))))
    (register-table
      (list (list 'pc pc) (list 'flag flag))))
  (define (allocate-register name)
    (if (assoc name register-table)
        (error "Multiply defined register: " name)
        (set! register-table
              (cons (list name (make-register name))
                    register-table)))
    'register-allocated)
  (define (lookup-register name)
    (let ((val (assoc name register-table)))
      (if val
          (cadr val)
          (error "Unknown register: " name))))
  (define (execute)
    (let ((insts (get-contents pc)))
      (if (null? insts)
          'done
          (begin
            (instruction-execution-proc (car insts))
            (set! the-instruction-counter (+ the-instruction-counter 1))
            (execute)))))
  (define (dispatch message)
    (cond ((eq? message 'start)
           (set-contents! pc the-instruction-sequence)
           (execute))
          ((eq? message 'install-instruction-sequence)
           (lambda (seq) (set! the-instruction-sequence seq)))
          ((eq? message 'allocate-register) allocate-register)
          ((eq? message 'get-register) lookup-register)
          ((eq? message 'install-operations)
           (lambda (ops) (set! the-ops (append the-ops ops))))
          ((eq? message 'stack) stack)
          ((eq? message 'operations) the-ops)
          ((eq? message 'counter) the-instruction-counter)
          ((eq? message 'reset-counter) (set! the-instruction-counter 0))
          (else (error "Unknown request -- MACHINE" message))))
    dispatch)))

```

```

(define factorial-machine
  (make-machine
    '(continue n val)
    (list (list '=) (list '* *) (list '- -))
    '(
      (perform (op initialize-stack))
      (assign continue (label factorial-done))

      factorial-loop
      (test (op =) (reg n) (const 0))
      (branch (label base-case))
      (test (op =) (reg n) (const 1))
      (branch (label base-case))
      (save continue)
      (save n)
      (assign continue (label after-factorial))
      (assign n (op -) (reg n) (const 1))
      (goto (label factorial-loop))
      after-factorial

      (restore n)
      (restore continue)
      (assign val (op *) (reg val) (reg n))
      (goto (reg continue))

      base-case
      (assign val (const 1))
      (goto (reg continue))

      factorial-done
      (perform (op print-stack-statistics))
    )))
)

(define (driver-loop)
  (newline)
  (let ((n (read)))
    (if (eq? n 'quit)
        'goodbye
        (begin (set-register-contents! factorial-machine 'n n)
               (start factorial-machine)

               (display "counter: ")
               (display (factorial-machine 'counter))
               (newline)

               (display "value: ")
               (display (get-register-contents factorial-machine 'val))
               (newline)

               (display "ratio of counter to n: ")
               (display (/ (/ (factorial-machine 'counter) 1.0) n))
               (newline)

               (factorial-machine 'reset-counter)

               (driver-loop)))))

  ;;;test data

5
total-pushes: 8
maximum-depth: 8
counter: 61
value: 120
ratio of counter to n: 12.2

10
total-pushes: 18
maximum-depth: 18
counter: 126
value: 3628800
ratio of counter to n: 12.6

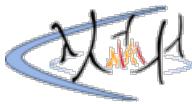
20
total-pushes: 38
maximum-depth: 38
counter: 256
value: 2432902008176640000
ratio of counter to n: 12.8

```

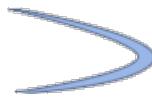
50  
total-pushes: 98  
maximum-depth: 98  
counter: 646  
value: 30414093201713378043612608166064768844377641568960512000000000000  
ratio of counter to n: 12.92

---

Last modified : 2020-01-29 06:02:54  
**WiLiKi 0.5-tekili-7** running on **Gauche 0.9**



# sicp-ex-5.16



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.15) | Index | Next exercise (5.17) >>

meteorgan

```
; add the code marked.
(define (make-new-machine)
  (let ((pc (make-register 'pc))
        (flag (make-register 'flag))
        (stack (make-stack))
        (the-instruction-sequence '()))
    (instruction-number 0)
    (trace-on false)) ; ***
(define (print-instruction-number)
  (display (list "current instruction number is: " instruction-number))
  (set! instruction-number 0)
  (newline))
(let ((the-ops
         (list (list 'initialize-stack
                     (lambda () (stack 'initialize)))
               (list 'print-stack-statistics
                     (lambda () (stack 'print-statistics)))))
        (register-table
         (list (list 'pc pc) (list 'flag flag))))
  (define (allocate-register name)
    (if (assoc name register-table)
        (error "Multiply defined register: " name)
        (set! register-table
              (cons (list name (make-register name))
                    register-table)))
    'register-allocated)
  (define (lookup-register name)
    (let ((val (assoc name register-table)))
      (if val
          (cadr val)
          (begin
            (allocate-register name)
            (lookup-register name)))))

(define (execute)
  (let ((insts (get-contents pc)))
    (if (null? insts)
        'done
        (begin
          (if trace-on ; ***
              (begin ; ***
                  (display (caar insts)) ; ***
                  (newline)) ; ***
                  ((instruction-execution-proc (car insts)))
                  (set! instruction-number (+ instruction-number 1))
                  (execute)))))

(define (dispatch message)
  (cond ((eq? message 'start)
         (set-contents! pc the-instruction-sequence)
         (execute))
        ((eq? message 'install-instruction-sequence)
         (lambda (seq) (set! the-instruction-sequence seq)))
        ((eq? message 'allocate-register) allocate-register)
        ((eq? message 'trace-on) (set! trace-on true)) ; ***
        ((eq? message 'trace-off) (set! trace-on false)) ; ***
        ((eq? message 'get-register) lookup-register)
        ((eq? message 'install-operations)
         (lambda (ops) (set! the-ops (append the-ops ops))))
        ((eq? 'instruction-number) print-instruction-number)
        ((eq? message 'stack) stack)
        ((eq? message 'operations) the-ops)
        (else (error "Unknown request -- MACHINE" message)))))

(dispatch))
(define (trace-on-instruction machine) ; ***
  (machine 'trace-on))
(define (trace-off-instruction machine) ; ***
  (machine 'trace-off))
```

codybartfast

Instead of having calls to *display* in the machine procedure (as the exercise asks) here's a variation that passes a message sink in as a parameter to trace-on!.

```
(define (make-new-machine)
  ...
  (let (( ... ))
    ...
    (define write-trace
      (lambda (message) '()))
    (define (trace-on sink)
      (set! write-trace sink))
    (define (trace-off)
      (set! write-trace (lambda (message) '())))
    (define (execute)
      (let ((insts (get-contents pc)))
        (if (null? insts)
            'done
            (begin
              (write-trace (caar insts)) ;; ***
              ((instruction-execution-proc (car insts)))
              (set! inst-count (+ inst-count 1))
              (execute)))))
    (define (dispatch message)
      ...
      (cond ((eq? message 'trace-on) trace-on)
            ((eq? message 'trace-off) trace-off)
            (else (error "Unknown request -- MACHINE" message)))
      dispatch))

  (define (trace-on! machine sink)
    ((machine 'trace-on) sink))

  (define (trace-off! machine)
    ((machine 'trace-off)))

Usage:
(trace-on! machine
  (lambda (message)
    (newline)
    (display "--trace--: ")
    (display message)))
```

revc

```
(define (make-stack)
  (let ((s '()))
    (number-pushes 0)
    (max-depth 0)
    (current-depth 0))
  (define (push x)
    (set! s (cons x s))
    (set! number-pushes (+ 1 number-pushes))
    (set! current-depth (+ 1 current-depth))
    (set! max-depth (max current-depth max-depth)))
  (define (pop)
    (if (null? s)
        (error "Empty stack -- POP" 'pop)
        (let ((top (car s)))
          (set! s (cdr s))
          (set! current-depth (- current-depth 1))
          top)))

  (define (initialize)
    (set! s '())
    (set! number-pushes 0)
    (set! max-depth 0)
    (set! current-depth 0)
    'done)
  (define (print-statistics)
    (newline)
    (for-each display (list "total-pushes: " number-pushes
                           "\n"
                           "maximum-depth: " max-depth
                           "\n"
                           )))

  (define (dispatch message)
    (cond ((eq? message 'push) push)
          ((eq? message 'pop) (pop))))
```

```

((eq? message 'initialize) (initialize))
((eq? message 'print-statistics)
  (print-statistics))
(else
  (error "Unknown request -- STACK" message)))
dispatch))

(define (make-new-machine)
  (let ((pc (make-register 'pc))
        (flag (make-register 'flag))
        (stack (make-stack))
        (the-instruction-sequence '()))
    (trace-switch #f))
  (let ((the-ops
         (list (list 'initialize-stack
                     (lambda () (stack 'initialize)))
               ;**next for monitored stack (as in section 5.2.4)
               ;; -- comment out if not wanted
               (list 'print-stack-statistics
                     (lambda () (stack 'print-statistics))))))
    (register-table
      (list (list 'pc pc) (list 'flag flag))))
  (define (allocate-register name)
    (if (assoc name register-table)
        (error "Multiply defined register: " name)
        (set! register-table
              (cons (list name (make-register name))
                    register-table)))
  'register-allocated)
  (define (lookup-register name)
    (let ((val (assoc name register-table)))
      (if val
          (cadr val)
          (error "Unknown register:" name))))
  (define (execute cnt)
    (let ((insts (get-contents pc)))
      (if (null? insts)
          'done
          (begin
            (if trace-switch
                (begin (display cnt)
                      (display ": \t")
                      (display (instruction-text (car insts)))
                      (newline)))
                ((instruction-execution-proc (car insts))
                 (execute (+ 1 cnt)))))))

  (define (dispatch message)
    (cond ((eq? message 'start)
           (set-contents! pc the-instruction-sequence)
           (execute 1))
          ((eq? message 'install-instruction-sequence)
           (lambda (seq) (set! the-instruction-sequence seq)))
          ((eq? message 'allocate-register) allocate-register)
          ((eq? message 'get-register) lookup-register)
          ((eq? message 'install-operations)
           (lambda (ops) (set! the-ops (append the-ops ops))))
          ((eq? message 'stack) stack)
          ((eq? message 'operations) the-ops)
          ((eq? message 'trace-on) (set! trace-switch #t))
          ((eq? message 'trace-off) (set! trace-switch #f))
          (else (error "Unknown request -- MACHINE" message))))
  dispatch)))

(define factorial-machine
  (make-machine
    '(continue n val)
    (list (list '= =) (list '* *) (list '- -))
    '(
      (perform (op initialize-stack))
      (assign continue (label factorial-done))

      factorial-loop
      (test (op =) (reg n) (const 0))
      (branch (label base-case))
      (test (op =) (reg n) (const 1))
      (branch (label base-case))
      (save continue)
      (save n)
      (assign continue (label after-factorial))
      (assign n (op -) (reg n) (const 1))
      (goto (label factorial-loop))
      after-factorial

      (restore n)
      (restore continue)

```

```

(assign val (op *) (reg val) (reg n))
(goto (reg continue))

base-case
(assign val (const 1))
(goto (reg continue))

factorial-done
(perform (op print-stack-statistics))
))

(factorial-machine 'trace-on)

(define input-prompt ";;; Factorial-Machine input:")
(define output-prompt ";;; Factorial-Machine output:")

(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))

(define (announce-output string)
  (newline) (display string) (newline))

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((n (read)))
    (announce-output output-prompt)
    (cond [(eq? n 'quit) (display "goodbye\n")]
          [(eq? n 'trace-on) (factorial-machine 'trace-on) (display "enable trace\n")]
          (driver-loop)
          [(eq? n 'trace-off) (factorial-machine 'trace-off) (display "disable trace\n")]
          (driver-loop)
          [(integer? n) (set-register-contents! factorial-machine 'n n)
           (start factorial-machine)
           (display "value: ")
           (display (get-register-contents factorial-machine 'val))
           (newline)
           (driver-loop)]
          [else (display "Unknown input, try again!\n") (driver-loop)])))
  )

;;;;;;;;;;;;;;
;;;;;test;;;;;;
;;;;; Factorial-Machine input:
1

;;;;; Factorial-Machine output:
1:   (perform (op initialize-stack))
2:   (assign continue (label factorial-done))
3:   (test (op =) (reg n) (const 0))
4:   (branch (label base-case))
5:   (test (op =) (reg n) (const 1))
6:   (branch (label base-case))
7:   (assign val (const 1))
8:   (goto (reg continue))
9:   (perform (op print-stack-statistics))

total-pushes: 0
maximum-depth: 0
value: 1

;;;;; Factorial-Machine input:
trace-off

;;;;; Factorial-Machine output:
disable trace

;;;;; Factorial-Machine input:
1

;;;;; Factorial-Machine output:

total-pushes: 0
maximum-depth: 0
value: 1

```



# sicp-ex-5.17

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (5.16) | Index | Next exercise (5.18) >>

meteorgan

```
; add a field to instruction to include label. and change the code in extract-labels
(define (make-instruction text)
  (list text '() '()))
(define (make-instruction-with-label text label)
  (list text label '()))
(define (instruction-text inst)
  (car inst))
(define (instruction-label inst)
  (cadr inst))
(define (instruction-execution-proc inst)
  (caddr inst))
(define (set-instruction-execution-proc! inst proc)
  (set-car! (cddr inst) proc))

(define (extract-labels text)
  (if (null? text)
      '()
      (let ((result (extract-labels (cdr text))))
        (let ((insts (car result)) (labels (cdr result)))
          (let ((next-inst (car text)))
            (if (symbol? next-inst)
                (if (label-exist? labels next-inst)
                    (error "the label has existed EXTRACT-LABELS" next-inst)
                    (let ((insts
                          (if (null? insts)
                              '()
                              (cons (make-instruction-with-label
                                     (instruction-text (car
insts))
                                     next-inst)
                                     (cdr insts)))))))
                  (cons insts
                        (cons (make-label-entry next-inst insts) labels))))
            (cons (cons (make-instruction next-inst) insts)
                  labels)))))))

;; change the code in execute in make-new-machine
(define (execute)
  (let ((insts (get-contents pc)))
    (if (null? insts)
        'done
        (begin
          (if trace-on
              (begin
                (if (not (null? (instruction-label (car insts))))
                    (begin
                      (display (instruction-label (car insts)))
                      (newline)))
                  (display (instruction-text (car insts)))
                  (newline)))
                ((instruction-execution-proc (car insts)))
                (set! instruction-number (+ instruction-number 1))
                (execute)))))))
```

Rptx

```
; This works but is less efficient than meteorgans answer because it
; searches for the label each time an instruction is executed.
; added a `labels` variable to `make-new-machine` with a message `install-labels`
; and a message `print-labels`.
; The assemble will pass the `install-labels` message in the receive procedure
; it passes to `extract-labels`.
; created a procedure `inst-label` which takes an instruction as argument and
; returns the label under which the instruction is.
; This is now printed by the execute procedure also.
; Each label has all the instruction starting from the one that follows it
```

```

; till the last one.

; this procedure is internal to make-new-machine
(define (inst-label inst)
  (define (inst-label-iter inst lst)
    (if (member inst (car lst))
        (caar lst)
        (inst-label-iter inst (cdr lst))))
  (inst-label-iter inst (reverse labels)))

; new messages in make-new-machine
((eq? message 'install-labels)
   (lambda (lbls) (set! labels lbls) 'done))
((eq? message 'print-labels)
   (lambda () labels))

; modified assemble procedure.
(define (assemble controller-text machine)
  (extract-labels controller-text
    (lambda (insts labels)
      ((machine 'install-labels) labels) ; install the labels.
      (update-insts! insts labels machine)
      insts)))

```

donald

```

;;my solution trace the "goto" inst and "branch" inst to update the current-label. need a little more work to store the first-label

;;add this at the begin of "make-new-machine" proc,
(current-label 'first-label)

;;update execute
(define (execute)
  (let ((insts (get-contents pc)))
    (if (null? insts)
        'done
        (let ((inst (car insts)))
          (begin (cond ((trace-on)
                        (display current-label) ;***
                        (newline)
                        (display (instruction-text inst))
                        (newline)))
                    ((instruction-execution-proc inst))
                    (set! instruction-number (+ instruction-number 1))
                    ;***
                    (if (or (tagged-list? (instruction-text inst) 'goto)
                            (and (tagged-list? (instruction-text inst) 'branch)
                                 (get-contents flag)))
                        (set! current-label
                              (label-exp-label (cadr (instruction-text)))))
                    ;***
                    (execute)))))))

```

codybartfast

I did this by adding the labels to the list of instructions. It works but it is a BAD solution as it effectively breaks the instruction list interface as each consumer of the instruction list now needs to check each instruction to see if it is a real instruction or just a label.

revc

```

(define (make-stack)
  (let ((s '()))
    (number-pushes 0)
    (max-depth 0)
    (current-depth 0))
  (define (push x)
    (set! s (cons x s))
    (set! number-pushes (+ 1 number-pushes))
    (set! current-depth (+ 1 current-depth))
    (set! max-depth (max current-depth max-depth)))
  (define (pop)
    (if (null? s)
        (error "Empty stack -- POP" 'pop)
        (let ((top (car s)))
          (set! s (cdr s))
          (set! current-depth (- current-depth 1))
          top)))

```

```

(define (initialize)
  (set! s '())
  (set! number-pushes 0)
  (set! max-depth 0)
  (set! current-depth 0)
  'done)
(define (print-statistics)
  (newline)
  (for-each display (list "total-pushes: " number-pushes
                           "\n"
                           "maximum-depth: " max-depth
                           "\n"
                           )))
(define (dispatch message)
  (cond ((eq? message 'push) push)
        ((eq? message 'pop) (pop))
        ((eq? message 'initialize) (initialize))
        ((eq? message 'print-statistics)
         (print-statistics))
        (else
         (error "Unknown request -- STACK" message))))
  dispatch))

(define input-prompt ";; Factorial-Machine input:")
(define output-prompt ";; Factorial-Machine output:")

(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))

(define (announce-output string)
  (newline) (display string) (newline))

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((n (read)))
    (announce-output output-prompt)
    (cond [(eq? n 'quit) (display "goodbye\n")]
          [(eq? n 'trace-on) (factorial-machine 'trace-on) (display "enable trace\n")]
          (driver-loop)]
          [(eq? n 'trace-off) (factorial-machine 'trace-off) (display "disable trace\n")]
          (driver-loop)
          [(integer? n)
           (set-register-contents! factorial-machine 'n n)
           (start factorial-machine)
           (display "value: ")
           (display (get-register-contents factorial-machine 'val))
           (newline)
           (driver-loop)]
          [else (display "Unknown input, try again!\n") (driver-loop)])))

(define (assemble controller-text machine)
  (extract-labels controller-text
    (lambda (insts labels)
      (update-insts! insts labels machine)
      ((machine 'install-instruction-labels) labels)
      insts))) ;**

(define (make-new-machine)
  (let ((pc (make-register 'pc))
        (flag (make-register 'flag))
        (stack (make-stack))
        (the-instruction-sequence '()))
    (the-instruction-labels '())
    (trace-switch #t))
  (let ((the-ops
         (list (list 'initialize-stack
                     (lambda () (stack 'initialize)))
               ;;**next for monitored stack (as in section 5.2.4)
               ;; -- comment out if not wanted
               (list 'print-stack-statistics
                     (lambda () (stack 'print-statistics))))))
    (register-table
      (list (list 'pc pc) (list 'flag flag))))
  (define (allocate-register name)
    (if (assoc name register-table)
        (error "Multiply defined register: " name)
        (set! register-table
              (cons (list name (make-register name))
                    register-table)))
  'register-allocated)
  (define (lookup-register name)
    (let ((val (assoc name register-table)))
      (if val
```

```

    (cadr val)
    (error "Unknown register:" name)))))

(define (lookup-insts labels insts)
  (let ((val (assoc insts labels)))
    (if val
        (cdr val)
        #f)))

(define (execute cnt)
  (let ((insts (get-contents pc)))
    (if (null? insts)
        'done
        (begin
          (if trace-switch
              (begin
                (let ((label (lookup-insts the-instruction-labels insts)))
                  (if label
                      (begin
                        (display "\t")
                        (display label)
                        (newline)))

                    (display cnt)
                    (display ": \t")
                    (display (instruction-text (car insts)))
                    (newline)))
                ((instruction-execution-proc (car insts)))
                (execute (+ 1 cnt))))))

      (define (dispatch message)
        (cond ((eq? message 'start)
               (set-contents! pc the-instruction-sequence)
               (execute 1))
              ((eq? message 'install-instruction-sequence)
               (lambda (seq) (set! the-instruction-sequence seq)))
              ((eq? message 'install-instruction-labels) ;**
               (lambda (labels) (set! the-instruction-labels (map
                                                               (lambda (x) (cons (cdr x)
                                                               (car x))) labels))))
               ((eq? message 'allocate-register) allocate-register)
               ((eq? message 'get-register) lookup-register)
               ((eq? message 'install-operations)
                (lambda (ops) (set! the-ops (append the-ops ops))))
               ((eq? message 'stack) stack)
               ((eq? message 'operations) the-ops)
               ((eq? message 'trace-on) (set! trace-switch #t))
               ((eq? message 'trace-off) (set! trace-switch #f))
               (else (error "Unknown request -- MACHINE" message))))
            dispatch)))

(define factorial-machine
  (make-machine
    '(continue n val)
    (list (list '= =) (list '* *) (list '- -))
    '(
      (perform (op initialize-stack))
      (assign continue (label factorial-done))

      factorial-loop
      (test (op =) (reg n) (const 0))
      (branch (label base-case))
      (test (op =) (reg n) (const 1))
      (branch (label base-case))
      (save continue)
      (save n)
      (assign continue (label after-factorial))
      (assign n (op -) (reg n) (const 1))
      (goto (label factorial-loop))
      after-factorial

      (restore n)
      (restore continue)
      (assign val (op *) (reg val) (reg n))
      (goto (reg continue))

      base-case
      (assign val (const 1))
      (goto (reg continue))

      factorial-done
      (perform (op print-stack-statistics))
    )))

```

```

;::::::::::::::::::test

;;; Factorial-Machine input:
1

;;; Factorial-Machine output:
1:   (perform (op initialize-stack))
2:   (assign continue (label factorial-done))
  factorial-loop
3:   (test (op =) (reg n) (const 0))
4:   (branch (label base-case))
5:   (test (op =) (reg n) (const 1))
6:   (branch (label base-case))
  base-case
7:   (assign val (const 1))
8:   (goto (reg continue))
  factorial-done
9:   (perform (op print-stack-statistics))

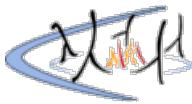
total-pushes: 0
maximum-depth: 0
value: 1

;;; Factorial-Machine input:
2

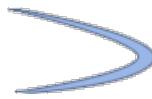
;;; Factorial-Machine output:
1:   (perform (op initialize-stack))
2:   (assign continue (label factorial-done))
  factorial-loop
3:   (test (op =) (reg n) (const 0))
4:   (branch (label base-case))
5:   (test (op =) (reg n) (const 1))
6:   (branch (label base-case))
7:   (save continue)
8:   (save n)
9:   (assign continue (label after-factorial))
10:  (assign n (op -) (reg n) (const 1))
11:  (goto (label factorial-loop))
  factorial-loop
12:  (test (op =) (reg n) (const 0))
13:  (branch (label base-case))
14:  (test (op =) (reg n) (const 1))
15:  (branch (label base-case))
  base-case
16:  (assign val (const 1))
17:  (goto (reg continue))
  after-factorial
18:  (restore n)
19:  (restore continue)
20:  (assign val (op *) (reg val) (reg n))
21:  (goto (reg continue))
  factorial-done
22:  (perform (op print-stack-statistics))

total-pushes: 2
maximum-depth: 2
value: 2

```



# sicp-ex-5.18



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.17) | Index | Next exercise (5.19) >>

meteorgan

```
(define (trace-on-register machine register-name)
  ((get-register machine register-name) 'trace-on)
  'trace-on)
(define (trace-off-register machine register-name)
  ((get-register machine register-name) 'trace-off)
  'trace-off)

(define (make-register name)
  (let ((contents '*unassigned*)
        (trace? false))
    (define (dispatch message)
      (cond ((eq? message 'get) contents)
            ((eq? message 'set)
             (lambda (value)
               (if trace?
                   (begin
                     (display name)
                     (display " ")
                     (display contents)
                     (display " ")
                     (display value)
                     (newline)
                     (set! contents value))
                   (set! contents value))))
            ((eq? message 'trace-on)
             (set! trace? true))
            ((eq? message 'trace-off)
             (set! trace? false))
            (else
              (error "Unknown request -- REGISTER" message)))))

    dispatch))
```

codybartfast

Instead of having calls to *display* in the register procedure (as the exercise asks) here's a variation that passes a message handler in as a parameter to trace-on!.

```
(define (write-null . message-parts) '())
(define (make-register name)
  (let ((contents '*unassigned*))
    (define write-trace write-null)
    (define (trace-on sink)
      (set! write-trace sink))
    (define (trace-off)
      (set! write-trace write-null))
    (define (dispatch message)
      (cond ((eq? message 'get) contents)
            ((eq? message 'set)
             (lambda (value)
               (write-trace name contents value)
               (set! contents value)))
            ((eq? message 'trace-on) trace-on)
            ((eq? message 'trace-off) trace-off)
            (else
              (error "Unknown request -- REGISTER" message)))))

    dispatch))
```

Usage:

```
(reg-trace-on! machine 'val
  (lambda (reg before after)
    (newline)
    (display reg)
    (display ": ")
    (display before))
```

```
(display " -> ")
(display after)))
```

```
revc
(define (make-stack)
  (let ((s '())
        (number-pushes 0)
        (max-depth 0)
        (current-depth 0))
    (define (push x)
      (set! s (cons x s))
      (set! number-pushes (+ 1 number-pushes))
      (set! current-depth (+ 1 current-depth))
      (set! max-depth (max current-depth max-depth)))
    (define (pop)
      (if (null? s)
          (error "Empty stack -- POP" 'pop)
          (let ((top (car s)))
            (set! s (cdr s))
            (set! current-depth (- current-depth 1))
            top)))
    (define (initialize)
      (set! s '())
      (set! number-pushes 0)
      (set! max-depth 0)
      (set! current-depth 0)
      'done)
    (define (print-statistics)
      (newline)
      (for-each display (list "total-pushes: " number-pushes
                               "\n"
                               "maximum-depth: " max-depth
                               "\n"
                               )))
    (define (dispatch message)
      (cond ((eq? message 'push) push)
            ((eq? message 'pop) (pop))
            ((eq? message 'initialize) (initialize))
            ((eq? message 'print-statistics)
             (print-statistics))
            (else
              (error "Unknown request -- STACK" message))))
      dispatch))
(define input-prompt ";; Factorial-Machine input:")
(define output-prompt ";; Factorial-Machine output:")

(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))

(define (announce-output string)
  (newline) (display string) (newline))

(define (lookup-insts labels insts)
  (let ((val (assoc insts labels)))
    (if val
        (cdr val)
        #f)))

(define (assemble controller-text machine)
  (extract-labels controller-text
    (lambda (insts labels)
      (update-insts! insts labels machine)
      ((machine 'install-instruction-labels) labels)
      insts)))

(define (make-register name labels)
  (let ((contents '*unassigned*)
        (trace-switch #f))

    (define (print-information value)
      (display "\n\n*****\n")
      (display ">>> ")
      (display "name:")
      (display name)
      (newline)
      (display ">>> "))

    (print-information value)
    (display "\n\n*****\n")))
```

```

(display "old: ")
(if (pair? contents)
    (if (lookup-insts labels contents)
        (display (lookup-insts labels contents))
        (pretty-print (list (instruction-text (car contents)) '...)))
    (display contents))
(newline)

(display ">>> ")
(display "new: ")
(if (pair? value)
    (if (lookup-insts labels value)
        (display (lookup-insts labels value))
        (pretty-print (list (instruction-text (car value)) '...)))
    (display value))

(newline)

(display "*****\n")
)

(define (dispatch message)
(cond ((eq? message 'get) contents)
((eq? message 'set)
(lambda (value)
(if trace-switch (print-information value))
(set! contents value)))
((eq? message 'trace-on) (set! trace-switch #t))
((eq? message 'trace-off) (set! trace-switch #f)))
(else
(error "Unknown request -- REGISTER" message)))
dispatch))

(define (make-new-machine)
(let* ((stack (make-stack)
(the-instruction-sequence '())
(the-instruction-labels '(*DUMMY* . *HEAD*)))
(flag (make-register 'flag the-instruction-labels))
(pc (make-register 'pc the-instruction-labels))
(trace-switch #f))
(let ((the-ops
(list (list 'initialize-stack
(lambda () (stack 'initialize)))
;**next for monitored stack (as in section 5.2.4)
;; -- comment out if not wanted
(list 'print-stack-statistics
(lambda () (stack 'print-statistics)))))
(register-table
(list (list 'pc pc) (list 'flag flag))))
(define (allocate-register name)
(if (assoc name register-table)
(error "Multiply defined register: " name)
(set! register-table
(cons (list name (make-register name the-instruction-labels)) ;**
register-table)))
'register-allocated)
(define (lookup-register name)
(let ((val (assoc name register-table)))
(if val
(cadr val)
(error "Unknown register:" name)))))

(define (execute cnt)
(let ((insts (get-contents pc)))
(if (null? insts)
'done
(begin
(if trace-switch
(begin
(let ((label (lookup-insts the-instruction-labels insts)))
(if label
(begin
(display "\t")
(display label)
(newline)))
(display cnt)
(display ":")
(display (instruction-text (car insts)))
(newline)))
((instruction-execution-proc (car insts)))
(execute (+ 1 cnt)))))

(define (dispatch message)
(cond ((eq? message 'start)
(set-contents! pc the-instruction-sequence)

```

```

        (execute 1))
((eq? message 'install-instruction-sequence)
  (lambda (seq) (set! the-instruction-sequence seq)))
((eq? message 'install-instruction-labels) ;**
  (lambda (labels) (set-cdr! the-instruction-labels (map
    (lambda (x) (cons (cdr x)
      (car x))) labels))))
  ((eq? message 'allocate-register) allocate-register)
  ((eq? message 'get-register) lookup-register)
  ((eq? message 'install-operations)
    (lambda (ops) (set! the-ops (append the-ops ops))))
  ((eq? message 'stack) stack)
  ((eq? message 'operations) the-ops)
  ((eq? message 'trace-on) (set! trace-switch #t))
  ((eq? message 'trace-off) (set! trace-switch #f))
  ((eq? message 'register-trace-off)
    (lambda (r) ((lookup-register r) 'trace-off)))
  ((eq? message 'register-trace-on)
    (lambda (r) ((lookup-register r) 'trace-on)))
  (else (error "Unknown request -- MACHINE" message)))
dispatch))

(define factorial-machine
  (make-machine
    '(continue n val)
    (list (list '= =) (list '* *) (list '- -))
    '(
      (perform (op initialize-stack))
      (assign continue (label factorial-done))

      factorial-loop
      (test (op =) (reg n) (const 0))
      (branch (label base-case))
      (test (op =) (reg n) (const 1))
      (branch (label base-case))
      (save continue)
      (save n)
      (assign continue (label after-factorial))
      (assign n (op -) (reg n) (const 1))
      (goto (label factorial-loop))
      after-factorial

      (restore n)
      (restore continue)
      (assign val (op *) (reg val) (reg n))
      (goto (reg continue))

      base-case
      (assign val (const 1))
      (goto (reg continue))

      factorial-done
      (perform (op print-stack-statistics))
    )))
)

(define (register-trace-off machine)
  (display "which register?\n")
  (let ((r (read)))
    ((machine 'register-trace-off) r)
    (display "disable trace of ")
    (display r)
    (newline)
    (driver-loop)))

(define (register-trace-on machine)
  (display "which register?\n")
  (let ((r (read)))
    ((machine 'register-trace-on) r)
    (display "enable trace of ")
    (display r)
    (newline)
    (driver-loop)))

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((n (read)))
    (announce-output output-prompt)
    (cond [(eq? n 'quit) (display "goodbye\n")]
          [(eq? n 'trace-on) (factorial-machine 'trace-on) (display "enable trace\n")]
    (driver-loop)]
          [(eq? n 'trace-off) (factorial-machine 'trace-off) (display "disable trace\n")]
    (driver-loop)]
          [(eq? n 'r-trace-off) (register-trace-off factorial-machine)]
          [(eq? n 'r-trace-on) (register-trace-on factorial-machine)])))

```

```

[ (integer? n)
  (set-register-contents! factorial-machine 'n n)
  (start factorial-machine)
  (display "value: ")
  (display (get-register-contents factorial-machine 'val))
  (newline)
  (driver-loop)]
[else (display "Unknown input, try again!\n") (driver-loop))]

;;;;;;;;;;;
;;TEST

> (driver-loop)

;;; Factorial-Machine input:
1

;;; Factorial-Machine output:

total-pushes: 0
maximum-depth: 0
value: 1

;;; Factorial-Machine input:
r-trace-on

;;; Factorial-Machine output:
which register?
continue
enable trace of continue

;;; Factorial-Machine input:
1

;;; Factorial-Machine output:

*****
>>> name:continue
>>> old: factorial-done
>>> new: factorial-done
*****


total-pushes: 0
maximum-depth: 0
value: 1

;;; Factorial-Machine input:
2

;;; Factorial-Machine output:

*****
>>> name:continue
>>> old: factorial-done
>>> new: factorial-done
*****


*****
>>> name:continue
>>> old: factorial-done
>>> new: after-factorial
*****


*****
>>> name:continue
>>> old: after-factorial
>>> new: factorial-done
*****


total-pushes: 2
maximum-depth: 2
value: 2

```

```

;;; Factorial-Machine input:
trace-on

;;; Factorial-Machine output:
enable trace

;;; Factorial-Machine input:
2

;;; Factorial-Machine output:
1:   (perform (op initialize-stack))
2:   (assign continue (label factorial-done))

*****
>>> name:continue
>>> old: factorial-done
>>> new: factorial-done
*****

factorial-loop
3:   (test (op =) (reg n) (const 0))
4:   (branch (label base-case))
5:   (test (op =) (reg n) (const 1))
6:   (branch (label base-case))
7:   (save continue)
8:   (save n)
9:   (assign continue (label after-factorial))

*****
>>> name:continue
>>> old: factorial-done
>>> new: after-factorial
*****

10:  (assign n (op -) (reg n) (const 1))
11:  (goto (label factorial-loop))
    factorial-loop
12:  (test (op =) (reg n) (const 0))
13:  (branch (label base-case))
14:  (test (op =) (reg n) (const 1))
15:  (branch (label base-case))
    base-case
16:  (assign val (const 1))
17:  (goto (reg continue))
    after-factorial
18:  (restore n)
19:  (restore continue)

*****
>>> name:continue
>>> old: after-factorial
>>> new: factorial-done
*****

20:  (assign val (op *) (reg val) (reg n))
21:  (goto (reg continue))
    factorial-done
22:  (perform (op print-stack-statistics))

total-pushes: 2
maximum-depth: 2
value: 2

;;; Factorial-Machine input:
r-trace-off

;;; Factorial-Machine output:
which register?
continue
disable trace of continue

;;; Factorial-Machine input:
2

;;; Factorial-Machine output:
1:   (perform (op initialize-stack))
2:   (assign continue (label factorial-done))
    factorial-loop
3:   (test (op =) (reg n) (const 0))
4:   (branch (label base-case))

```

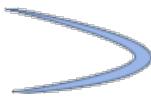
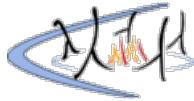
```
5:      (test (op =) (reg n) (const 1))
6:      (branch (label base-case))
7:      (save continue)
8:      (save n)
9:      (assign continue (label after-factorial))
10:     (assign n (op -) (reg n) (const 1))
11:     (goto (label factorial-loop))
12:     factorial-loop
13:     (test (op =) (reg n) (const 0))
14:     (branch (label base-case))
15:     (test (op =) (reg n) (const 1))
16:     (branch (label base-case))
17:     base-case
18:     (assign val (const 1))
19:     (goto (reg continue))
20:     after-factorial
21:     (restore n)
22:     (restore continue)
23:     (assign val (op *) (reg val) (reg n))
24:     (goto (reg continue))
25:     factorial-done
26:     (perform (op print-stack-statistics))

total-pushes: 2
maximum-depth: 2
value: 2
```

```
;;; Factorial-Machine input:
quit
```

```
;;; Factorial-Machine output:
goodbye
```

# sicp-ex-5.19



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.18) | Index | Next exercise (5.20) >>

meteorgan

```
; using a list to contain break points, every element in the list
; is a pair containing the label and line. the code changed
;; had been marked.

(define (make-new-machine)
  (let ((pc (make-register 'pc))
        (flag (make-register 'flag))
        (stack (make-stack))
        (the-instruction-sequence '()))
    (instruction-number 0)
    (trace-on false)
    (labels '())
    (current-label '*unassigned*) ;; ***
    (current-line 0) ;; ***
    (breakpoint-line 0) ;; ***
    (break-on true)) ;; ****

  (let ((the-ops
         (list (list 'initialize-stack
                     (lambda () (stack 'initialize)))
               (list 'print-stack-statistics
                     (lambda () (stack 'print-statistics))))
         (register-table
          (list (list 'pc pc) (list 'flag flag))))
  (define (print-instruction-number)
    (display (list "current instruction number is: " instruction-number))
  (set! instruction-number 0)
  (newline))
  (define (allocate-register name)
    (if (assoc name register-table)
        (error "Multiply defined register: " name)
        (set! register-table
              (cons (list name (make-register name))
                    register-table)))
    'register-allocated)
  (define (lookup-register name)
    (let ((val (assoc name register-table)))
      (if val
          (cadr val)
          (begin
            (allocate-register name)
            (lookup-register name)))))
  (define (execute)
    (let ((insts (get-contents pc)))
      (if (null? insts)
          'done
          (begin
            (if (and (not (null? (instruction-label (car insts)))) ;; ***
                     (assoc (instruction-label (car insts)) labels))
                (begin
                  (set! current-label (instruction-label (car insts)))
                  (set! breakpoint-line (cdr (assoc current-label labels)))
                  (set! current-line 0))
                (set! current-line (+ current-line 1))
                (if (and (= current-line breakpoint-line) break-on)
                    (begin
                      (set! break-on false)
                      (display (list "breakpoint here" current-label current-line)))
                    (begin
                      (if trace-on
                          (begin
                            (display (instruction-text (car insts)))
                            (newline)))
                      (set! break-on true)))) ;;
            ((instruction-execution-proc (car insts)))
            (set! instruction-number (+ instruction-number 1))
            (execute)))))))
  (define (cancel-breakpoint label) ;; ***
  (define (delete-label acc-labels orig-labels)
    (cond ((null? orig-labels)
           (error "the label is not in the machine -- CANCEL-BREAKPOINT"))
          (else (delete-label (cons label acc-labels) orig-labels))))
```

```

label))
      ((eq? (caar orig-labels) label) (append acc-labels (cdr orig-labels)))
          (else (delete-label (cons (car orig-labels) acc-labels) (cdr orig-
labels))))
  (set! labels (delete-label '() labels))
  (define (dispatch message)
    (cond ((eq? message 'start)
           (set-contents! pc the-instruction-sequence)
           (execute))
          ((eq? message 'install-instruction-sequence)
           (lambda (seq) (set! the-instruction-sequence seq)))
          ((eq? message 'allocate-register) allocate-register)
          ((eq? message 'trace-on) (set! trace-on true))
          ((eq? message 'trace-off) (set! trace-on false))
          ((eq? message 'get-register) lookup-register)
          ((eq? message 'install-operations)
           (lambda (ops) (set! the-ops (append the-ops ops))))
          ((eq? message 'instruction-number) print-instruction-number)
          ((eq? message 'stack) stack)
          ((eq? message 'operations) the-ops)
          ((eq? message 'set-breakpoint) ;; ***
           (lambda (label n) (set! labels (cons (cons label n) labels))))
          ((eq? message 'cancel-breakpoint) ;; ***
           (lambda (label) (cancel-breakpoint label)))
          ((eq? message 'cancel-all-breakpoint) (set! labels '()))
          ((eq? message 'process-machine) (execute)) ;; ***
          (else (error "Unknown request -- MACHINE" message)))
  dispatch))

(define (set-breakpoint machine label n)
  ((machine 'set-breakpoint) label n))
(define (cancel-breakpoint machine label)
  ((machine 'cancel-breakpoint) label))
(define (cancel-all-breakpoint machine)
  (machine 'cancel-all-breakpoint))
(define (process-machine machine)
  (machine 'process-machine))

```

Another solution at:

<https://github.com/spacemanakai/sicp/blob/master/ch5/ex-5.19-breakpoints.scm>

(codybartfast: This may have moved to:

[https://github.com/michiakig/sicp/blob/master/ch5/ex-5.19-breakpoints.scm\)](https://github.com/michiakig/sicp/blob/master/ch5/ex-5.19-breakpoints.scm)

poly

I think the above procedure will function well with only one breakpoint, but there might be some bug when implement it with more than one breakpoint.

```

...
(if (and (not (null? (instruction-label (car insts))))
         (assoc (instruction-label (car insts)) labels))
  (begin
    (set! current-label (instruction-label (car insts)))
    (set! breakpoint-line (cdr (assoc current-label labels)))
    (set! current-line 0)))
...

```

the (set! breakpoint-line (cdr (assoc current-label labels))) here will cause some bug if there are two breakpoints with same label in the labels (labels are used to store breakpoints here). Apparently, the front one in the labels will be calculated first. But actually both of them should be calculated simultaneously.

The second situation is that if two breakpoints are interleaved. For example, if breakpoint one is set up to break when the procedure meet label "A" after 100 steps, and breakpoint two to label "B" after 50 steps. It will be alright if label "B" is not in the 100 steps after label "A". But if it is in, the above procedure will just cover the counting current-line and breakpoint-line, reset the current-line to 0 and breakpoint-line to 50, which means process won't be broke for breakpoint one (A, 100).

aos

I ended up using a 2-D association list as my breakpoint table. The first key is the label, the second key is the line number, and the third key is whether it is on or off. Setting a breakpoint adds (or modifies) the breakpoint in the table to on, canceling turns it off. Unsetting all BPs just resets the table completely.

In this way you can interleave breakpoints to your heart's desire.

codybartfast

I took a different approach and updated the instruction object to indicate if we should break on that instruction and which records the original label & offset. At runtime the machine then

just checks if the instruction is marked as a breakpoint.

```
;; The breakpoint-controller provides procedures for modifying instructions:  
  
(define (make-breakpoint-controller labels)  
  (define (set label offset)  
    (set-instruction-break!  
      (list-ref (lookup-label labels label) offset)  
      #t (cons label offset)))  
  (define (cancel label offset)  
    (set-instruction-break!  
      (list-ref (lookup-label labels label) offset) #f '()))  
  (define (cancel-all)  
    (map  
      (lambda (label)  
        (map  
          (lambda (inst)  
            (set-instruction-break! inst #f '()))  
          (filter  
            (lambda (inst) (not (symbol? inst)))  
            (cdr label))))  
      labels))  
  (define (dispatch message)  
    (cond  
      ((eq? message 'set) set)  
      ((eq? message 'cancel) cancel)  
      ((eq? message 'cancel-all) cancel-all)))  
  dispatch)  
  
;; And in the machine:  
  
(define (execute-proceed check-break)  
  (let ((insts (get-contents pc)))  
    (cond ((null? insts)  
           ...  
           ((and check-break (instruction-break? (car insts)))  
            (let ((desc (instruction-break-desc (car insts))))  
              (display "--break--: label: ")  
              (display (car desc))  
              (display " offset: ")  
              (display (cdr desc))  
              (newline)  
              'stopped))  
           (else ;; normal execution  
            (write-trace (instruction-text (car insts)))  
            ((instruction-execution-proc (car insts)))  
            (set! inst-count (+ inst-count 1))  
            (execute))))  
  (define (proceed) (execute-proceed #false))  
  (define (execute) (execute-proceed #true))  
  
;; full code on github:  
;; https://github.com/codybartfast/sicp/blob/master/chapter5/exercise-5.19.scm  
;; https://github.com/codybartfast/sicp/blob/master/chapter5/machine-19.scm
```

revc

```
(define (make-stack)  
  (let ((s '())  
        (number-pushes 0)  
        (max-depth 0)  
        (current-depth 0))  
  (define (push x)  
    (set! s (cons x s))  
    (set! number-pushes (+ 1 number-pushes))  
    (set! current-depth (+ 1 current-depth))  
    (set! max-depth (max current-depth max-depth)))  
  (define (pop)  
    (if (null? s)  
        (error "Empty stack -- POP" 'pop)  
        (let ((top (car s)))  
          (set! s (cdr s))  
          (set! current-depth (- current-depth 1))  
          top)))  
  
  (define (initialize)  
    (set! s '())  
    (set! number-pushes 0)  
    (set! max-depth 0)  
    (set! current-depth 0)  
    'done)  
  (define (print-statistics)  
    (newline))
```

```

(for-each display (list "total-pushes: " number-pushes
                        "\n"
                        "maximum-depth: " max-depth
                        "\n"
                        )))

(define (dispatch message)
  (cond ((eq? message 'push) push)
        ((eq? message 'pop) (pop))
        ((eq? message 'initialize) (initialize))
        ((eq? message 'print-statistics)
         (print-statistics))
        (else
         (error "Unknown request -- STACK" message))))
  dispatch))

(define input-prompt ";; Factorial-Machine input:")
(define output-prompt ";; Factorial-Machine output:")

(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))

(define (announce-output string)
  (newline) (display string) (newline))

(define (lookup-insts labels insts)
  (let ((val (assoc insts labels)))
    (if val
        (cdr val)
        #f)))

(define (assemble controller-text machine)
  (extract-labels controller-text
    (lambda (insts labels)
      (update-insts! insts labels machine)
      ((machine 'install-instruction-labels) labels)
      insts)))

(define (make-register name labels)
  (let ((contents '*unassigned*)
        (trace-switch #f))

    (define (print-information value)
      (display "register ")
      (display name)

      (display " :")
      (if (pair? contents)
          (if (lookup-insts labels contents)
              (display (lookup-insts labels contents))
              (display (list (instruction-text (car contents)) '...)))
          (display contents))

      (display " >>> ")
      (if (pair? value)
          (if (lookup-insts labels value)
              (display (lookup-insts labels value))
              (display (list (instruction-text (car value)) '...)))
          (display value))

      (newline))

    (define (dispatch message)
      (cond ((eq? message 'get) contents)
            ((eq? message 'set)
             (lambda (value)
               (if trace-switch (print-information value))
               (set! contents value)))
            ((eq? message 'trace-on) (set! trace-switch #t))
            ((eq? message 'trace-off) (set! trace-switch #f))
            (else
             (error "Unknown request -- REGISTER" message))))
      dispatch))

(define (make-new-machine)
  (let* ((stack (make-stack))
         (the-instruction-sequence '())
         (the-instruction-insts '((*DUMMY* . *HEAD*)))
         (the-instruction-labels '((*DUMMY* . *HEAD*)))
         (the-breakpoints '())
         (flag (make-register 'flag the-instruction-insts))
         (pc (make-register 'pc the-instruction-insts)))

```

```

        (trace-switch #f)
(let ((the-ops
      (list (list 'initialize-stack
                  (lambda () (stack 'initialize)))
            ;;**next for monitored stack (as in section 5.2.4)
            ;; -- comment out if not wanted
            (list 'print-stack-statistics
                  (lambda () (stack 'print-statistics)))))

(register-table
  (list (list 'pc pc) (list 'flag flag))))
(define (allocate-register name)
  (if (assoc name register-table)
      (error "Multiply defined register: " name)
      (set! register-table
            (cons (list name (make-register name the-instruction-insts)) ;**
                  register-table)))
  'register-allocated)
(define (lookup-register name)
  (let ((val (assoc name register-table)))
    (if val
        (cadr val)
        (error "Unknown register: " name)))

(define (print-trace insts)
  (let ((label (lookup-insts the-instruction-insts insts)))
    (if label
        (begin
          (display "label:\t")
          (display label)
          (newline)))
        (display "instruction:\t")
        (display (instruction-text (car insts)))
        (newline)))

(define (proceed)
  (let ((insts (get-contents pc)))
    (if (null? insts)
        'done
        (begin
          (if trace-switch
              (print-trace insts))
          ((instruction-execution-proc (car insts)))
          (execute)))))

(define (execute)
  (let ((insts (get-contents pc)))
    (if (or (null? insts) (member insts the-breakpoints))
        'done
        (begin
          (if trace-switch
              (print-trace insts))
          ((instruction-execution-proc (car insts)))
          (execute)))))

(define (install-instruction-labels labels)
  (set-cdr! the-instruction-insts
            (map (lambda (x) (cons (cdr x) (car x)))
                 labels))
  (set-cdr! the-instruction-labels labels))

(define (set-breakpoint label n)
  (set! the-breakpoints
        (cons (list-tail (lookup-label the-instruction-labels label) n)
              the-breakpoints)))

(define (cancel-breakpoint label n)
  (set! the-breakpoints
        (remove (list-tail (lookup-label the-instruction-labels label) n)
                the-breakpoints)))

(define (cancel-all-breakpoints)
  (set! the-breakpoints '()))

(define (dispatch message)
  (cond ((eq? message 'start)
         (set-contents! pc the-instruction-sequence)
         (execute))
        ((eq? message 'install-instruction-sequence)
         (lambda (seq) (set! the-instruction-sequence seq)))
        ((eq? message 'install-instruction-labels) ;**
         install-instruction-labels)
        ((eq? message 'allocate-register) allocate-register)
        ((eq? message 'get-register) lookup-register)
        ((eq? message 'install-operations)
         (lambda (ops) (set! the-ops (append the-ops ops))))
        ((eq? message 'stack) stack)))

```

```

(((eq? message 'operations) the-ops)
((eq? message 'trace-on) (set! trace-switch #t))
((eq? message 'trace-off) (set! trace-switch #f))
((eq? message 'register-trace-off)
  (lambda (r) ((lookup-register r) 'trace-off)))
((eq? message 'register-trace-on)
  (lambda (r) ((lookup-register r) 'trace-on)))

  ((eq? message 'set-breakpoint) set-breakpoint)
  ((eq? message 'cancel-breakpoint) cancel-breakpoint)
  ((eq? message 'cancel-all-breakpoints) (cancel-all-breakpoints))
  ((eq? message 'proceed-machine) (proceed))
  (else (error "Unknown request -- MACHINE" message))))
dispatch))

(define factorial-machine
  (make-machine
    '(continue n val)
    (list (list '= =) (list '* *) (list '- -))
    '(
      (perform (op initialize-stack))
      (assign continue (label factorial-done))

      factorial-loop
      (test (op =) (reg n) (const 0))
      (branch (label base-case))
      (test (op =) (reg n) (const 1))
      (branch (label base-case))
      (save continue)
      (save n)
      (assign continue (label after-factorial))
      (assign n (op -) (reg n) (const 1))
      (goto (label factorial-loop))
      after-factorial

      (restore n)
      (restore continue)
      (assign val (op *) (reg val) (reg n))
      (goto (reg continue))

      base-case
      (assign val (const 1))
      (goto (reg continue))

      factorial-done
      (perform (op print-stack-statistics))
    )))

(define (register-trace-off machine)
  (display "which register?\n")
  (let ((r (read)))
    ((machine 'register-trace-off) r)
    (display "disable trace of ")
    (display r)
    (newline)
    (driver-loop)))

(define (register-trace-on machine)
  (display "which register?\n")
  (let ((r (read)))
    ((machine 'register-trace-on) r)
    (display "enable trace of ")
    (display r)
    (newline)
    (driver-loop)))

(define (breakpoint-setter machine)
  (display "which label?\n")
  (let ((label (read)))
    (display "which position?\n")
    (let ((n (read)))
      (set-breakpoint machine label n)
      (driver-loop)))

(define (breakpoint-canceler machine)
  (display "which label?\n")
  (let ((label (read)))
    (display "which position?\n")
    (let ((n (read)))
      (cancel-breakpoint machine label n)
      (driver-loop)))

(define (driver-loop)
  (prompt-for-input input-prompt))

```

```

(let ((n (read)))
  (announce-output output-prompt)
  (cond [(eq? n 'quit) (display "goodbye\n")]
        [(eq? n 'trace-on) (factorial-machine 'trace-on) (display "enable trace\n")]
(drive-loop)
        [(eq? n 'trace-off) (factorial-machine 'trace-off) (display "disable trace\n")]
(drive-loop)
        [(eq? n 'r-trace-off) (register-trace-off factorial-machine)]
        [(eq? n 'r-trace-on) (register-trace-on factorial-machine)]
        [(eq? n 'set-bp) (breakpoint-setter factorial-machine)]
        [(eq? n 'cancel-bp) (breakpoint-canceler factorial-machine)]
        [(eq? n 'cancel-abp) (cancel-all-breakpoints factorial-machine) (drive-loop)]
        [(eq? n 'proceed) (proceed-machine factorial-machine) (drive-loop)]
        [(integer? n)
         (set-register-contents! factorial-machine 'n n)
         (start factorial-machine)
         (display "value: ")
         (display (get-register-contents factorial-machine 'val))
         (newline)
         (drive-loop)]
        [else (display "Unknown input, try again!\n") (drive-loop)))))

(define (set-breakpoint machine label n)
  ((machine 'set-breakpoint) label n))

(define (cancel-breakpoint machine label n)
  ((machine 'cancel-breakpoint) label n))

(define (cancel-all-breakpoints machine)
  (machine 'cancel-all-breakpoints))

(define (proceed-machine machine)
  (machine 'proceed-machine))

;;;;;;;;;;test;;;;;;;;;;;

;;; Factorial-Machine input:
trace-on

;;; Factorial-Machine output:
enable trace

;;; Factorial-Machine input:
r-trace-on

;;; Factorial-Machine output:
which register?
continue
enable trace of continue

;;; Factorial-Machine input:
2

;;; Factorial-Machine output:
instruction: (perform (op initialize-stack))
instruction: (assign continue (label factorial-done))
register continue :*unassigned* >>> factorial-done
label: factorial-loop
instruction: (test (op =) (reg n) (const 0))
instruction: (branch (label base-case))
instruction: (test (op =) (reg n) (const 1))
instruction: (branch (label base-case))
instruction: (save continue)
instruction: (save n)
instruction: (assign continue (label after-factorial))
register continue :factorial-done >>> after-factorial
instruction: (assign n (op -) (reg n) (const 1))
instruction: (goto (label factorial-loop))
label: factorial-loop
instruction: (test (op =) (reg n) (const 0))
instruction: (branch (label base-case))
instruction: (test (op =) (reg n) (const 1))
instruction: (branch (label base-case))
label: base-case
instruction: (assign val (const 1))
instruction: (goto (reg continue))
label: after-factorial
instruction: (restore n)
instruction: (restore continue)
register continue :after-factorial >>> factorial-done
instruction: (assign val (op *) (reg val) (reg n))
instruction: (goto (reg continue))
label: factorial-done
instruction: (perform (op print-stack-statistics))

```

```
total-pushes: 2
maximum-depth: 2
value: 2

;;; Factorial-Machine input:
set-bp

;;; Factorial-Machine output:
which label?
factorial-done
which position?
0

;;; Factorial-Machine input:
1

;;; Factorial-Machine output:
instruction: (perform (op initialize-stack))
instruction: (assign continue (label factorial-done))
register continue :factorial-done >>> factorial-done
label: factorial-loop
instruction: (test (op =) (reg n) (const 0))
instruction: (branch (label base-case))
instruction: (test (op =) (reg n) (const 1))
instruction: (branch (label base-case))
label: base-case
instruction: (assign val (const 1))
instruction: (goto (reg continue))
value: 1

;;; Factorial-Machine input:
proceed

;;; Factorial-Machine output:
label: factorial-done
instruction: (perform (op print-stack-statistics))

total-pushes: 0
maximum-depth: 0

;;; Factorial-Machine input:
set-bp

;;; Factorial-Machine output:
which label?
base-case
which position?
0

;;; Factorial-Machine input:
1

;;; Factorial-Machine output:
instruction: (perform (op initialize-stack))
instruction: (assign continue (label factorial-done))
register continue :factorial-done >>> factorial-done
label: factorial-loop
instruction: (test (op =) (reg n) (const 0))
instruction: (branch (label base-case))
instruction: (test (op =) (reg n) (const 1))
instruction: (branch (label base-case))
value: 1

;;; Factorial-Machine input:
proceed

;;; Factorial-Machine output:
label: base-case
instruction: (assign val (const 1))
instruction: (goto (reg continue))

;;; Factorial-Machine input:
proceed

;;; Factorial-Machine output:
label: factorial-done
instruction: (perform (op print-stack-statistics))

total-pushes: 0
```

```

maximum-depth: 0

;;; Factorial-Machine input:
cancel-bp

;;; Factorial-Machine output:
which label?
factorial-done
which position?
0

;;; Factorial-Machine input:
1

;;; Factorial-Machine output:
instruction: (perform (op initialize-stack))
instruction: (assign continue (label factorial-done))
register continue :factorial-done >>> factorial-done
label: factorial-loop
instruction: (test (op =) (reg n) (const 0))
instruction: (branch (label base-case))
instruction: (test (op =) (reg n) (const 1))
instruction: (branch (label base-case))
value: 1

;;; Factorial-Machine input:
proceed

;;; Factorial-Machine output:
label: base-case
instruction: (assign val (const 1))
instruction: (goto (reg continue))
label: factorial-done
instruction: (perform (op print-stack-statistics))

total-pushes: 0
maximum-depth: 0

;;; Factorial-Machine input:
cancel-abp

;;; Factorial-Machine output:

;;; Factorial-Machine input:
1

;;; Factorial-Machine output:
instruction: (perform (op initialize-stack))
instruction: (assign continue (label factorial-done))
register continue :factorial-done >>> factorial-done
label: factorial-loop
instruction: (test (op =) (reg n) (const 0))
instruction: (branch (label base-case))
instruction: (test (op =) (reg n) (const 1))
instruction: (branch (label base-case))
label: base-case
instruction: (assign val (const 1))
instruction: (goto (reg continue))
label: factorial-done
instruction: (perform (op print-stack-statistics))

total-pushes: 0
maximum-depth: 0
value: 1

;;; Factorial-Machine input:
quit

;;; Factorial-Machine output:
goodbye

```

# sicp-ex-5.20

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

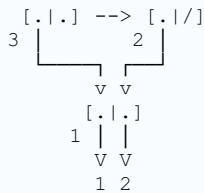
<< Previous exercise (5.19) | Index | Next exercise (5.21) >>

aos

Here's my answer to this one!

```
; -----> x <-----
; |-----| 1 | 2 |
; |-----|
; | 3 |
; |-----|
; | • | • | --> | • | / |
; |-----|
; 1           2
;
;          0   1   2   3   4
;
; the-cars |   | p3 | p3 | n1 |   |
;-----|
; the-cdrs |   | p2 | e0 | n2 |   |
;-----|
```

codybartfast



Index 0 1 2 3 4 5

the-cars		n1	p1	p1		
the-cdrs		n2	e0	p2		

```
free = p4
x    = p1
y    = p3
```

Looking at various answers on the interweb there seems to be little agreement on the order in which values are allocated to memory. A common variation is:

Index 0 1 2 3 4 5

the-cars		n1	p1	p1		
the-cdrs		n2	p3	e0		

This doesn't look right to me because the address that (p1 . e0) is stored in is not known until after (p1 . e0) is saved to memory:

```
(perform
  (op vector-set!) (reg the-cars) (reg free) (reg <reg2>))
(perform
  (op vector-set!) (reg the-cdrs) (reg free) (reg <reg3>))
  (assign <reg1> (reg free)) ;-- address not available until here
  (assign free (op +) (reg free) (const 1))
```

So the contents of the inner cons must already be stored before the outer

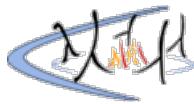
cons can use the address of the inner cons as its cdr.

Generally, when looking at Lisp expressions, I would imagine values are stored in the order that operations complete, not in the order that they are called.

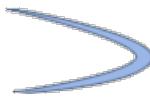
**revc**

Box-and-pointer and memory-vector representations:

**here**



# sicp-ex-5.21



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.20) | Index | Next exercise (5.22) >>

meteorgan

```
(a)
> (define count-leaves-machine
  (make-machine
    (list (list '+ +) (list 'null? null?)
          (list 'pair? pair?) (list 'car car) (list 'cdr cdr))
    '(
      (assign continue (label count-leaves-done))
      (assign val (const 0))
    tree-loop
      (test (op null?) (reg tree))
      (branch (label null-tree))
      (test (op pair?) (reg tree))
      (branch (label left-tree))
      (assign val (const 1))
      (goto (reg continue)))
    left-tree
      (save tree)
      (save continue)
      (assign continue (label right-tree))
      (assign tree (op car) (reg tree))
      (goto (label tree-loop)))
    right-tree
      (restore continue)
      (restore tree)
      (save continue)
      (save val)
      (assign continue (label after-tree))
      (assign tree (op cdr) (reg tree))
      (goto (label tree-loop)))
    after-tree
      (assign var (reg val))
      (restore val)
      (restore continue)
      (assign val (op +) (reg var) (reg val))
      (goto (reg continue)))
    null-tree
      (assign val (const 0))
      (goto (reg continue)))
    count-leaves-done)))

(set-register-contents! count-leaves-machine 'tree '(a (b c (d)) (e f) g))
(start count-leaves-machine)
(get-register-contents count-leaves-machine 'val)
'done
'done
7
```

Rptx

```
; a.

(define (not-pair? lst)
  (not (pair? lst)))

(define count-leaves
  (make-machine
    `((car ,car) (cdr ,cdr) (null? ,null?))
    (not-pair? ,not-pair?) (+ ,+))
  '(
    start
      (assign continue (label done))
      (assign n (const 0))
    count-loop
      (test (op null?) (reg lst))
      (branch (label null))
      (test (op not-pair?) (reg lst))
      (branch (label not-pair))
```

```

(save continue)
(assign continue (label after-car))
(save lst)
(assign lst (op car) (reg lst))
(goto (label count-loop))
after-car
  (restore lst)
  (assign lst (op cdr) (reg lst))
  (assign continue (label after-cdr))
  (save val)
  (goto (label count-loop))
after-cdr
  (restore n)
  (restore continue)
  (assign val
         (op +) (reg val) (reg n))
  (goto (reg continue))
null
  (assign val (const 0))
  (goto (reg continue))
not-pair
  (assign val (const 1))
  (goto (reg continue))
done)))

```

**; ; b.**

```

(define count-leaves
  (make-machine
    `((car ,car) (cdr ,cdr) (pair? ,pair?)
      (null? ,null?) (+ ,+))
    '(
      start
        (assign val (const 0))
        (assign continue (label done))
        (save continue)
        (assign continue (label cdr-loop))
      count-loop
        (test (op pair?) (reg lst))
        (branch (label pair))
        (test (op null?) (reg lst))
        (branch (label null))
        (assign val (op +) (reg val) (const 1))
        (restore continue)
        (goto (reg continue))
      cdr-loop
        (restore lst)
        (assign lst (op cdr) (reg lst))
        (goto (label count-loop))
      pair
        (save lst)
        (save continue)
        (assign lst (op car) (reg lst))
        (goto (label count-loop))
      null
        (restore continue)
        (goto (reg continue))
      done)))

```

donald

**; ;b)**

```

(define c-m
  (make-machine '()
    (list (list 'null? null?)
          (list 'pair? pair?)
          (list '+ +)
          (list 'car car)
          (list 'cdr cdr)))
    '(controller
      (assign n (const 0))
      (assign continue (label iter-done)))

    iter
      (test (op null?) (reg tree))
      (branch (label null-tree))
      (test (op pair?) (reg tree))
      (branch (label pair-tree))
      (assign n (op +) (reg n) (const 1))
      (goto (reg continue))

    null-tree
      (goto (reg continue)))

```

```

pair-tree
(save continue)
(save tree)
(assign tree (op car) (reg tree))
(assign continue (label after-left-tree))
(goto (label iter))

after-left-tree
	restore tree)
(assign tree (op cdr) (reg tree))
(assign continue (label after-right-tree))
(goto (label iter))

after-right-tree
	restore continue)
(goto (reg continue))

iter-done)))

```

### revc

```

(define count-leaves-machine
  (make-machine
    '(continue counter aux tree)
    (list (list 'null? null?)
          (list 'integer? integer?)
          (list 'symbol? symbol?)
          (list '+ +)
          (list 'car car)
          (list 'cdr cdr)))

    '(
      (assign continue (label cl-done))
      cl-loop
      (test (op null?) (reg tree))
      (branch (label null-case))
      (test (op integer?) (reg tree))
      (branch (label atom-case))
      (test (op symbol?) (reg tree))
      (branch (label atom-case))

      (save continue)
      (save tree)
      (assign continue (label after-cl-1))
      (assign tree (op car) (reg tree))
      (goto (label cl-loop))

      after-cl-1
      (restore tree)
      (assign tree (op cdr) (reg tree))
      (save counter)
      (assign continue (label after-cl-2))
      (goto (label cl-loop))

      after-cl-2
      (restore aux)
      (assign counter (op +) (reg counter) (reg aux))

      (restore continue)
      (goto (reg continue))

      null-case
      (assign counter (const 0))
      (goto (reg continue))

      atom-case
      (assign counter (const 1))
      (goto (reg continue))

      cl-done
    )))
  (set-register-contents! count-leaves-machine 'tree '(1 (3 4) 5 (6 (7 3) 9)))
  (start count-leaves-machine)
  (printf "~a~%" (get-register-contents count-leaves-machine 'counter))

  (define count-leaves-machine
    (make-machine
      '(n tree source-tree continue)
      (list (list 'null? null?)
            (list 'integer? integer?)))

```

```

(list 'symbol? symbol?)
(list '+ +)
(list 'car car)
(list 'cdr cdr))
'(
  (assign continue (label cl-done))

  (assign tree (reg source-tree))
  (assign n (const 0))

  cl-loop
  (test (op null?) (reg tree))
  (branch (label null-case))
  (test (op integer?) (reg tree))
  (branch (label atom-case))
  (test (op symbol?) (reg tree))
  (branch (label atom-case))

  (save continue)
  (save tree)
  (assign continue (label after-cl-1))
  (assign tree (op cdr) (reg tree))
  (goto (label cl-loop))

  after-cl-1
  (restore tree)
  (assign tree (op car) (reg tree))
  (assign continue (label after-cl-2))
  (goto (label cl-loop))

  after-cl-2
  (restore continue)
  (goto (reg continue))

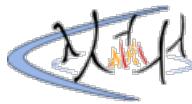
  null-case
  (goto (reg continue))

  atom-case
  (assign n (op +) (reg n) (const 1))
  (goto (reg continue))

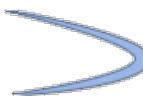
  cl-done
  )))

(set-register-contents! count-leaves-machine 'source-tree '(1 (3 4) 5 (6 (7 3) 9)))
(start count-leaves-machine)
(sprintf "~a~%" (get-register-contents count-leaves-machine 'n))

```



# sicp-ex-5.22



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.21) | Index | Next exercise (5.23) >>

Rptx

```
; a. append

(define append-machine
  (make-machine
    `((null? ,null?) (cons ,cons) (car ,car)
      (cdr ,cdr)))
  '(
    start
      (assign x (reg x)) ; these 2 instruction are only here to
      (assign y (reg y)) ; initialize the registers.
      (assign continue (label done)) ; retnr address
      (save continue) ; save it.
    append
      (test (op null?) (reg x))
      (branch (label null))
      (assign temp (op car) (reg x)) ; push car as the arg to cons.
      (save temp)
      (assign continue (label after-rec)) ;return address for procedure call.
      (save continue) ; push the return address
      (assign x (op cdr) (reg x)) ; arg for recursive call to append.
      (goto (label append)) ; recursive call to append.
    after-rec
      (restore x) ; get the argument pushed by append
      (assign val (op cons) (reg x) (reg val)) ; consit to the return value
      (restore continue) ; get the return address
      (goto (reg continue)) ; return to caller.
    null
      (assign val (reg y)) ; base case, return value = y.
      (restore continue) ; get return address
      (goto (reg continue)) ; return to caller.
    done)))
  )

; b. append!

(define append!-machine
  (make-machine
    `((set-cdr! ,set-cdr!) (null? ,null?)
      (cdr ,cdr)))
  '(
    start
      (assign x (reg x)) ; as before just initiaile the regs.
      (assign y (reg y))
      (assign temp1 (reg x)) ; must use temp to avoid changing x.
      (goto (label last-pair))
    append
      (assign temp (op set-cdr!) (reg temp1) (reg y)) ;set-cdr! returns an
      (goto (label done)) ; unspecified value, that we put in temp.
    last-pair
      (assign temp (op cdr) (reg temp1)) ; test if (cdr temp1 is null)
      (test (op null?) (reg temp)) ; if so, temp1 is the last pair.
      (branch (label null))
      (assign temp1 (op cdr) (reg temp1))
    null
      (goto (label append!)) ; splice the lists.
    done
  ))
```

donald

```
; ;a)
(define append-m
  (make-machine '()
    (list (list 'null? null?))
```

```

        (list 'cdr cdr)
        (list 'car car)
        (list 'cons cons))
    ' (controller
      (assign continue (label append-done))

      loop
      (test (op null?) (reg x))
      (branch (label null-x))
      (save continue)
      (assign continue (label cdr-done))
      (assign car-x (op car) (reg x))
      (save car-x)
      (assign x (op cdr) (reg x))
      (goto (label loop))

      null-x
      (assign x (reg y))
      (goto (reg continue))

      cdr-done
      (restore car-x)
      (assign x (op cons) (reg car-x) (reg x))
      (restore continue)
      (goto (reg continue))

      append-done)))

```

**; ;b)**

```

(define append-m!
  (make-machine '()
    (list (list 'null? null?)
          (list 'cdr cdr)
          (list 'set-cdr! set-cdr!))
    ' (controller
      (assign iter-x (reg x))

      iter
      (assign cdr-x (op cdr) (reg iter-x))
      (test (op null?) (reg cdr-x))
      (branch (label do-append))
      (assign iter-x (op cdr) (reg iter-x))
      (goto (label iter))

      do-append
      (perform (op set-cdr!) (reg iter-x) (reg y)))))


```

revc

```

(define append-machine
  (make-machine
    '(result former latter continue)
    (list (list 'null? null?)
          (list 'integer? integer?)
          (list 'symbol? symbol?))
    (list '+ +)
    (list 'car car)
    (list 'cdr cdr)
    (list 'cons cons))
  ' (
    (assign continue (label append-done))

    append-loop
    (test (op null?) (reg former))
    (branch (label base-case))

    (save continue)
    (save former)
    (assign continue (label after-append))
    (assign former (op cdr) (reg former))
    (goto (label append-loop))

    after-append
    (restore former)
    (restore continue)
    (assign former (op car) (reg former))
    (assign result

```

```

        (op cons) (reg former) (reg result))

        (goto (reg continue))

    base-case
    (assign result (reg latter))
    (goto (reg continue))

    append-done
  )))

(set-register-contents! append-machine 'former '(1 2 4 3))
(set-register-contents! append-machine 'latter '(8 9 10))
(start append-machine)
(printf "~a~%" (get-register-contents append-machine 'result))

(define (append! x y)
  (if (null? (cdr x))
      (set-cdr! x y)
      (append! (cdr x) y)))

(define append!-machine
  (make-machine
    '(former latter aux-former rest)
    (list (list 'null? null?)
          (list 'integer? integer?)
          (list 'symbol? symbol?))
          (list '+ +)
          (list 'car car)
          (list 'cdr cdr)
          (list 'cons cons)
          (list 'set-cdr! set-cdr!)))
  '(
    (assign aux-former (reg former))
    (assign rest (op cdr) (reg aux-former))

    test-rest
    (test (op null?) (reg rest))
    (branch (label append!-done))

    (assign aux-former (op cdr) (reg aux-former))
    (assign rest (op cdr) (reg aux-former))
    (goto (label test-rest))

    append!-done
    (perform (op set-cdr!) (reg aux-former) (reg latter))
  ))

(set-register-contents! append!-machine 'former '(1 2 4 3))
(set-register-contents! append!-machine 'latter '(8 9 10))
(start append!-machine)
(printf "~a~%" (get-register-contents append!-machine 'former))

```

# sicp-ex-5.23

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (5.22) | Index | Next exercise (5.24) >>

meteorgan

```
; add those following ev-dispatch
(test (op cond?) (reg expr))
(branch (label ev-cond))

ev-cond
(assign expr (op cond->if) (reg expr))
(goto (label ev-if))

;; add those to eval-operations
(list 'cond? cond?)
(list 'cond->if cond->if)
```

aos

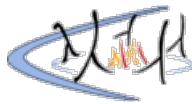
And for let, we can do something similar:

```
; Add to eval-dispatch
(test (op let?) (reg exp))
(branch (label ev-let))

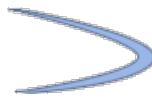
;; add this to eval operations
ev-let
(assign exp (op let->combination) (reg exp))
(goto (label ev-lambda))
```

let->combination was **exercise 4.6**

Last modified : 2019-06-23 23:36:58  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-5.24



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.23) | Index | Next exercise (5.25) >>

meteorgan

```
ev-cond
(assign expr (op cond-clauses) (reg expr))
(test (op null?) (reg expr))
(branch (label ev-cond-end))
(assign unev (op car) (reg expr))
(assign expr (op cdr) (reg expr))
(test (op cond-else-clauses?) (reg unev))
(branch (label cond-else))
(save env)
(save continue)
(save unev)
(save expr)
(assign continue (label ev-cond-loop))
(assign expr (op cond-predicate) (reg unev))
(goto (label ev-dispatch))

ev-cond-loop
(restore expr)
(test (op true?) (reg val))
(branch (label cond-result))
(restore unev)
(restore continue)
(restore env)
(goto (label ev-cond))

; this does not restore continue so it wont return to the caller.
; it also leaves env on the stack which would accumulate with
; each call to a cond.
cond-result
(restore unev)
(assign expr (op cond-actions) (reg unev))
(assign expr (op sequence->exp) (reg expr))
(goto (label ev-dispatch))

cond-else
(assign unev (op cond-actions) (reg unev))
(assign expr (op sequence->exp) (reg unev))
(goto (label ev-dispatch))

ev-cond-end
(goto (reg continue))
```

Rptx

This is a shorter version.

```
ev-cond
(save continue) ; save continue for ev-sequence
(assign unev (op cond-clauses) (reg exp))
ev-cond-loop
(test (op null?) (reg unev)) ; no more clauses
(branch (label ev-cond-unspec))
(assign exp (op cond-first-clause-predicate) (reg unev))
(test (op cond-else-predicate?) (reg exp))
(branch (label ev-cond-true))
(save unev) ; save clauses
(save env) ; and env to evaluate each clause
(assign continue (label ev-cond-decide))
(goto (label eval-dispatch)) ; eval first-predicate
ev-cond-decide
(restore env) ; get the env
(restore unev) ; get the clauses
(test (op true?) (reg val)) ; if predicate evaluates to true
(branch (label ev-cond-true)) ; goto ev-cond-true
(assign unev (op cond-rest-clauses) (reg unev))
(goto (label ev-cond-loop))
```

```

    ev-cond-true                                ; we found a true clause same as before
      (assign unev (op cond-first-clause-actions) (reg unev))
      (goto (label ev-sequence))                ; go to ev-sequence.

; if there was no else, and no true clause
; you could also leave val to be false here.
; But in the implementations I tested
; it was unspecified, or void.

    ev-cond-unspec
      (assign val (const 'unspecified)) ; assign val unspecified
      (restore continue)              ; go directly to caller.
      (goto (reg continue))

```

codybartfast

This is a longer version.

```

ev-cond
  (save continue)                                ; save final destination
  (assign exp (op cond-clauses) (reg exp))       ; drop cond label
ev-cond-have-clause?
  (test (op have-clause?) (reg exp))            ; any clauses?
  (branch (label ev-cond-check-clause))          ;      --> check clause
  (goto (label ev-cond-no-clauses))               ;      --> no clauses

ev-cond-check-clause
  (assign unev (op clauses-first) (reg exp))     ; get first clause
  (test (op cond-else-clause?) (reg unev))        ; else clause?
  (branch (label ev-cond-else))                  ;      --> else
  (save exp)                                     ; save clauses list
  (save unev)                                    ; save clause
  (save env)                                     ; save env
  (assign exp (op cond-predicate) (reg unev))    ; get predicate
  (assign continue (label ev-cond-after-predicate))
  (goto (label eval-dispatch))                  ;      --> eval predicate

ev-cond-after-predicate
  (restore env)                                 ; restore env
  (restore unev)                                ; restore clause
  (restore exp)                                 ; restore clauses list
  (test (op true?) (reg val))                  ; is predicate true?
  (branch (label ev-cond-actions))             ;      --> actions
  (assign exp (op clauses-rest) (reg exp))     ; drop first clause
  (goto (label ev-cond-have-clause?))           ;      --> try-again

ev-cond-else
  (assign val (op clauses-rest) (reg exp))     ; get clauses after else
  (test (op have-clause?) (reg val))           ; any clauses after else?
  (branch (label ev-cond-error-else-not-last)) ;      --> prepare error

ev-cond-actions
  (assign unev (op cond-actions) (reg unev))   ; store actions for ev-seq
  (goto (label ev-sequence))                   ;      --> ev-sequence

ev-cond-no-clauses
  (restore continue)                           ; restore continue
  (assign exp (const false))                  ; name of false variable
  (goto (label ev-variable))                 ;      --> lookup false value

ev-cond-error-else-not-last
  (restore (reg continue))                   ; restore continue
  (assign val (const else-not-last-clause--COND))
  (goto (label signal-error))               ;      --> raise error

```

No Match: if there's no match I believe #f should be returned, because:

- 1) that is what's returned by the metacircular evaluator,
- 2) that is what eceval returns for an if without an alternate clause.

Else Last Error: to match the behaviour of the metacircular evaluator, checks if the else clause is the last clause.

Sample

=====

```
(define (on-dice? n)
  (cond ((< n 1) false)
        (else (< n 7))))
```

Returns first clause:

-----

```
(cond
```

```

((on-dice? 1) "Hello from First Clause")
((on-dice? 2) "Hello from Second Clause")
(else "Hello from Else Clause"))

Output: eceval DONE - val: Hello from First Clause

Returns other clause:
-----
(cond
  ((on-dice? 9) "Hello from First Clause")
  ((on-dice? 2) "Hello from Second Clause")
  (else "Hello from Else Clause"))

Output: eceval DONE - val: Hello from Second Clause

Returns else clause:
-----
(cond
  ((on-dice? 0) "Hello from First Clause")
  ((on-dice? 7) "Hello from Second Clause")
  (else "Hello from Else Clause")))

Output: eceval DONE - val: Hello from Else Clause

Returns false if no match:
-----
(cond
  ((on-dice? 0) "Hello from First Clause")
  ((on-dice? 7) "Hello from Second Clause"))

Output: eceval DONE - val: #f

Error if else is not last clause
-----
(cond
  ((on-dice? 0) "Hello from First Clause")
  (else "Hello from Else Clause")
  ((on-dice? 2) "Hello from Third Clause"))

Output: ERROR: ELSE-clause-isnt-last--COND

```

revc

```

;;; Exercise 5.24
ev-cond
(save continue)
;; extract the clauses
(assign unev (op cond-clauses) (reg exp))
;; set default value for cond
(assgin val (const '<void>'))

ev-cond-loop
;; test for empty clauses
(test (op empty-clauses?) (reg unev))
(branch (label ev-cond-return))

;; extract the first
(assign exp (op first-clause) (reg unev))
(test (op cond-else-clause?) (reg exp))
(branch (label ev-cond-actions))

;; save the first
(save exp)
;; save the clauses
(save unev)
(save env)

(assign exp (op cond-predicate) (reg exp))
(assign continue (label ev-predicate-decide))
(goto (label eval-dispatch))

ev-predicate-decide
	restore env
	restore unev

```

```
(restore exp)
(test (op true?) (reg val))
(branch (label ev-cond-actions))

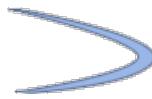
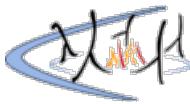
(assign unev (op rest-clauses) (reg unev))
(goto (label ev-cond-loop))

ev-cond-actions
(assign unev (op cond-actions) (reg exp))
; (save continue) ignored, since we have saved it before.
(goto (label ev-sequence))

ev-cond-return
(restore continue)
(goto (reg continue))
```

---

Last modified : 2020-02-04 04:26:39  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



<< Previous exercise (5.24) | Index | Next exercise (5.26) >>

aos

My solution can be found here:  
<https://github.com/-aos/SICP/blob/master/exercises/ch5/4/ex-25.ss>

revc

```
; ;;;EXPLICIT-CONTROL EVALUATOR FROM SECTION 5.4 OF
; ;;; STRUCTURE AND INTERPRETATION OF COMPUTER PROGRAMS

; ;;;Matches code in ch5.scm

; ;;; To use it
; ;;; -- load "load-eceval.scm", which loads this file and the
; ;;;     support it needs (including the register-machine simulator)

; ;;; -- To initialize and start the machine, do

; ;;; (define the-global-environment (setup-environment))

; ;;; (start ecleval)

; ;;; To restart, can do just
; ;;; (start ecleval)
; ;;;;;;

; ;;;**NB. To [not] monitor stack operations, comment in/[out] the line after
; ;;; print-result in the machine controller below
; ;;;**Also choose the desired make-stack version in regsim.scm

(define ecleval-operations
  (list
    ;;primitive Scheme operations
    (list 'read read)

    ;;operations in syntax.scm
    (list 'self-evaluating? self-evaluating?)
    (list 'quoted? quoted?)
    (list 'text-of-quotation text-of-quotation)
    (list 'variable? variable?)
    (list 'assignment? assignment?)
    (list 'assignment-variable assignment-variable)
    (list 'assignment-value assignment-value)
    (list 'definition? definition?)
    (list 'definition-variable definition-variable)
    (list 'definition-value definition-value)
    (list 'lambda? lambda?)
    (list 'lambda-parameters lambda-parameters)
    (list 'lambda-body lambda-body)
    (list 'if? if?)
    (list 'if-predicate if-predicate)
    (list 'if-consequent if-consequent)
    (list 'if-alternative if-alternative)
    (list 'begin? begin?)
    (list 'begin-actions begin-actions)
    (list 'last-exp? last-exp?)
    (list 'first-exp first-exp)
    (list 'rest-exp? rest-exp)
    (list 'application? application?)
    (list 'operator operator)
    (list 'operands operands)
    (list 'no-operands? no-operands?)
    (list 'first-operand first-operand)
    (list 'rest-operands rest-operands)
    ;; support cond
    (list 'cond? cond?)
    (list 'cond->if cond->if)

    ;;operations in eceval-support.scm
```

```

(list 'true? true?)
(list 'make-procedure make-procedure)
(list 'compound-procedure? compound-procedure?)
(list 'procedure-parameters procedure-parameters)
(list 'procedure-body procedure-body)
(list 'procedure-environment procedure-environment)
(list 'extend-environment extend-environment)
(list 'lookup-variable-value lookup-variable-value)
(list 'set-variable-value! set-variable-value!)
(list 'define-variable! define-variable!)
(list 'primitive-procedure? primitive-procedure?)
(list 'apply-primitive-procedure apply-primitive-procedure)
(list 'prompt-for-input prompt-for-input)
(list 'announce-output announce-output)
(list 'user-print user-print)
(list 'empty-arglist empty-arglist)
(list 'adjoin-arg adjoin-arg)
(list 'last-operand? last-operand?)
(list 'no-more-exp? no-more-exp?) ;for non-tail-recursive machine
(list 'get-global-environment get-global-environment)

;; the operations of thunk
(list 'delay-it delay-it)
(list 'thunk? thunk?)
(list 'evaluated-thunk? evaluated-thunk?)
(list 'thunk-exp thunk-exp)
(list 'thunk-env thunk-env)
(list 'thunk-value thunk-value)
(list 'set-car! set-car!)
(list 'set-cdr! set-cdr!)
(list 'cdr cdr)
(list 'car car)
(list 'pretty-print pretty-print)
(list 'display display)
)
)

(define ecleval
  (make-machine
    '(exp env val proc argl continue unev)
    ecleval-operations
    '(
      ;;SECTION 5.4.4
      read-eval-print-loop
        (perform (op initialize-stack))
        (perform
          (op prompt-for-input) (const ";; LAZY-Eval input:"))
        (assign exp (op read))
        (assign env (op get-global-environment))
        (assign continue (label print-result))
        (goto (label actual-value)) ;**
      print-result
      ;;**following instruction optional -- if use it, need monitored stack
      (perform (op print-stack-statistics))
      (perform
        (op announce-output) (const ";; LAZY-Eval value:"))
      (perform (op user-print) (reg val))
      (goto (label read-eval-print-loop))

      unknown-expression-type
      (assign val (const unknown-expression-type-error))
      (goto (label signal-error))

      unknown-procedure-type
      (restore continue)
      (assign val (const unknown-procedure-type-error))
      (goto (label signal-error))

      signal-error
      (perform (op user-print) (reg val))
      (goto (label read-eval-print-loop))

      ;;SECTION 5.4.1
      eval-dispatch
      (test (op self-evaluating?) (reg exp))
      (branch (label ev-self-eval))
      (test (op variable?) (reg exp))
      (branch (label ev-variable))
      (test (op quoted?) (reg exp))
      (branch (label ev-quoted))
      (test (op assignment?) (reg exp))
      (branch (label ev-assignment))
      (test (op definition?) (reg exp))
      (branch (label ev-definition))
      (test (op if?) (reg exp))
      (branch (label ev-if))
      (test (op cond?) (reg exp))
      )
    )
  )

```

```

(branch (label ev-cond))
(test (op lambda?) (reg exp))
(branch (label ev-lambda))
(test (op begin?) (reg exp))
(branch (label ev-begin))
(test (op application?) (reg exp))
(branch (label ev-application))
(goto (label unknown-expression-type))

ev-self-eval
  (assign val (reg exp))
  (goto (reg continue))
ev-variable
  (assign val (op lookup-variable-value) (reg exp) (reg env))
  (goto (reg continue))
ev-quoted
  (assign val (op text-of-quotation) (reg exp))
  (goto (reg continue))
ev-lambda
  (assign unev (op lambda-parameters) (reg exp))
  (assign exp (op lambda-body) (reg exp))
  ;; (perform (op pretty-print) (reg continue)) ;TEST
  (assign val (op make-procedure)
    (reg unev) (reg exp) (reg env))
  ;; (perform (op pretty-print) (reg continue)) ;TEST
  (goto (reg continue))

ev-application           ;EMPTY OPERANDS TODO
  (save continue)
  (save env)
  (assign unev (op operands) (reg exp))
  (save unev)
  (assign exp (op operator) (reg exp))
  (assign continue (label ev-appl-did-operator))
  (goto (label actual-value))      ;**

ev-appl-did-operator
  (restore unev)
  (restore env)

  (assign argl (op empty-arglist))
  (assign proc (reg val)) ;**
  (test (op no-operands?) (reg unev))
  (branch (label apply-dispatch))

  (test (op compound-procedure?) (reg proc))
  (branch (label compound-loop))      ;**

  (save proc)
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-loop))      ;**

  (restore proc)
  (goto (label unknown-procedure-type))

;;;
primitive-loop

  (save argl)

  (assign exp (op first-operand) (reg unev))

  (test (op last-operand?) (reg unev))
  (branch (label primitive-last-arg))

  (save env)
  (save unev)
  (assign continue (label primitive-accumulate-arg))
  (goto (label actual-value))

primitive-accumulate-arg

  (restore unev)
  (restore env)
  (restore argl)

  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (assign unev (op rest-operands) (reg unev))

  (goto (label primitive-loop))

primitive-last-arg
  (assign continue (label primitive-accum-last-arg))
  (goto (label actual-value))

primitive-accum-last-arg
  (restore argl)

```

```

(assign argl (op adjoin-arg) (reg val) (reg argl))
	restore proc
	goto (label primitive-apply)

primitive-apply
	(assign val (op apply-primitive-procedure)
		(reg proc)
		(reg argl))
	restore continue
	goto (reg continue))

;;;**
compound-loop
	(assign exp (op first-operand) (reg unev))
	(test (op last-operand?) (reg unev))
	(branch (label compound-accum-last-arg))

compound-accumulate-arg
	(assign val (op delay-it) (reg exp) (reg env)) ;**
	(assign argl (op adjoin-arg) (reg val) (reg argl))
	(assign unev (op rest-operands) (reg unev))
	goto (label compound-loop)

compound-accum-last-arg
	(assign val (op delay-it) (reg exp) (reg env)) ;**
	(assign argl (op adjoin-arg) (reg val) (reg argl))
	goto (label compound-apply))

compound-apply
	(assign unev (op procedure-parameters) (reg proc))
	(assign env (op procedure-environment) (reg proc))
	(assign env (op extend-environment)
		(reg unev) (reg argl) (reg env))
	(assign unev (op procedure-body) (reg proc))
	goto (label ev-sequence))

apply-dispatch
	(test (op primitive-procedure?) (reg proc))
	(branch (label primitive-apply))
	(test (op compound-procedure?) (reg proc))
	(branch (label compound-apply))
	(goto (label unknown-procedure-type))

actual-value
	(save continue)
	(assign continue (label after-eval))
	goto (label eval-dispatch))

after-eval
	(assign continue (label after-force))
	goto (label force-it))

after-force
	restore continue
	goto (reg continue))

force-it
	(test (op thunk?) (reg val))
	(branch (label ev-thunk))

	(test (op evaluated-thunk?) (reg val))
	(branch (label ev-evaluated-thunk))

	(goto (reg continue))

ev-thunk
	(save continue)
	(save val)

	(assign env (op thunk-env) (reg val))
	(assign exp (op thunk-exp) (reg val))
	(assign continue (label after-thunk))
	goto (label actual-value))

after-thunk
;; the register exp stores the thunk
	restore exp
	restore continue
	(perform (op set-car!) (reg exp) (const evaluated-thunk))
	(assign exp (op cdr) (reg exp))
	(perform (op set-car!) (reg exp) (reg val))
	(perform (op set-cdr!) (reg exp) (const ()))
	goto (reg continue))

ev-evaluated-thunk
	(assign val (op thunk-value) (reg val))
	goto (reg continue))

```

```

;;;SECTION 5.4.2
ev-begin
  (assign unev (op begin-actions) (reg exp))
  (save continue)
  (goto (label ev-sequence))

ev-sequence
  (assign exp (op first-exp) (reg unev))
  (test (op last-exp?) (reg unev))
  (branch (label ev-sequence-last-exp))
  (save unev)
  (save env)
  (assign continue (label ev-sequence-continue))
  (goto (label eval-dispatch))
ev-sequence-continue
  (restore env)
  (restore unev)
  (assign unev (op rest-exps) (reg unev))
  (goto (label ev-sequence))
ev-sequence-last-exp
  (restore continue)
  (goto (label eval-dispatch))

;;;SECTION 5.4.3

ev-if
  (save exp)
  (save env)
  (save continue)
  (assign continue (label ev-if-decide))
  (assign exp (op if-predicate) (reg exp))
  (goto (label eval-dispatch))
ev-if-decide
  (restore continue)
  (restore env)
  (restore exp)
  (test (op true?) (reg val))
  (branch (label ev-if-consequent))
ev-if-alternative
  (assign exp (op if-alternative) (reg exp))
  (goto (label eval-dispatch))
ev-if-consequent
  (assign exp (op if-consequent) (reg exp))
  (goto (label eval-dispatch))

ev-assignment
  (assign unev (op assignment-variable) (reg exp))
  (save unev)
  (assign exp (op assignment-value) (reg exp))
  (save env)
  (save continue)
  (assign continue (label ev-assignment-1))
  (goto (label eval-dispatch))
ev-assignment-1
  (restore continue)
  (restore env)
  (restore unev)
  (perform
    (op set-variable-value!) (reg unev) (reg val) (reg env))
  (assign val (const ok))
  (goto (reg continue)))

ev-definition
  (assign unev (op definition-variable) (reg exp))
  (save unev)
  (assign exp (op definition-value) (reg exp))
  (save env)
  (save continue)
  (assign continue (label ev-definition-1))
  (goto (label eval-dispatch))
ev-definition-1
  (restore continue)
  (restore env)
  (restore unev)
  (perform
    (op define-variable!) (reg unev) (reg val) (reg env))
  (assign val (const ok))
  (goto (reg continue)))

;;; Exercise 5.23
ev-cond
  (assign exp (op cond->if) (reg exp))
  (goto (label eval-dispatch))
))

'(EXPLICIT CONTROL LAZY EVALUATOR LOADED)

```

```

;;;;;;;;;;;
;;;;;test;;;;;;
;;;;;;;;;

;The procedures that prepare for the test
(define (cons x y)
  (lambda (m) (m x y)))

(define (car z)
  (z (lambda (p q) p)))

(define (cdr z)
  (z (lambda (p q) q)))

(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1)))))

(define (map proc items)
  (if (null? items)
      '()
      (cons (proc (car items))
            (map proc (cdr items))))))

(define (scale-list items factor)
  (map (lambda (x) (* x factor))
       items))

(define (add-lists list1 list2)
  (cond ((null? list1) list2)
        ((null? list2) list1)
        (else (cons (+ (car list1) (car list2))
                    (add-lists (cdr list1) (cdr list2))))))

(define ones (cons 1 ones))

(define integers (cons 1 (add-lists ones integers)))

(define (f) 1)

(define (g x y) (+ x y))

;;;;;;;;;; the test results

;;; LAZY-Eval input:
(f)

(total-pushes = 5 maximum-depth = 5)
;;; LAZY-Eval value:
1

;;; LAZY-Eval input:
(g 1 2)

(total-pushes = 22 maximum-depth = 10)
;;; LAZY-Eval value:
3

;;; LAZY-Eval input:
(list-ref integers 20)

(total-pushes = 3833 maximum-depth = 217)
;;; LAZY-Eval value:
21

;;; LAZY-Eval input:
(list-ref integers 20)

(total-pushes = 1166 maximum-depth = 211)
;;; LAZY-Eval value:
21

;;; LAZY-Eval input:
(car integers)

(total-pushes = 22 maximum-depth = 8)
;;; LAZY-Eval value:
1

;;; LAZY-Eval input:
(car (cdr integers))

(total-pushes = 43 maximum-depth = 15)

```

```

;;; LAZY-Eval value:
2

;;; LAZY-Eval input:
(+ 1 1)

(total-pushes = 12 maximum-depth = 7)
;;; LAZY-Eval value:
2

;;; LAZY-Eval input:
(+)

(total-pushes = 5 maximum-depth = 5)
;;; LAZY-Eval value:
0

;;; LAZY-Eval input:
(+ 1 1 1 1)

(total-pushes = 20 maximum-depth = 7)
;;; LAZY-Eval value:
4

;;; LAZY-Eval input:
(g 2 3)

(total-pushes = 22 maximum-depth = 10)
;;; LAZY-Eval value:
5

```

closeparen

We can avoid duplicating the whole structure of list-of-arg-values and list-of-delayed-values by borrowing the "proc" register to switch between actual-value and delay-it mode, depending on whether this is a primitive or compound procedure.

Solution expressed here as a diff from 5.24.

```

;; Parse error: Spurious closing paren found
--- 5.24.scm      2023-05-11 22:49:45.000000000 -0700
+++ 5.25.scm      2023-05-14 13:05:19.000000000 -0700
@@ -26,6 +26,7 @@
 (load "ch5-syntax.scm")
 (load "ch5-eceval-support.scm")

+;; let support
(define (let? exp) (tagged-list? exp 'let))
(define (let-bindings exp) (cadr exp))
(define (let-body exp) (caddr exp))
@@ -38,6 +39,21 @@
     (make-lambda (bindings->names (let-bindings exp)) (list (let-body exp)))
     (bindings->values (let-bindings exp)))

+;; thunk support
+
+(define (delay-it exp env)
+  (list 'thunk exp env))
+
+(define (thunk? obj)
+  (tagged-list? obj 'thunk))
+
+(define (thunk-exp thunk)
+  (cadr thunk))
+
+(define (thunk-env thunk)
+  (caddr thunk))
+
+

(define eceval-operations
  (list
@@ -108,6 +124,11 @@
    (list 'cdr cdr)
    (list 'sequence->exp sequence->exp))

+  (list 'delay-it delay-it)
+  (list 'thunk? thunk?))
+  (list 'thunk-exp thunk-exp))
+  (list 'thunk-env thunk-env))
+
  (list 'let? let?))
  (list 'let->lambda let->lambda)

```

```

        ))
@@ -125,7 +146,7 @@
    (assign exp (op read))
    (assign env (op get-global-environment))
    (assign continue (label print-result))
-   (goto (label eval-dispatch))
+   (goto (label ev-actual-value))
print-result
; ;**following instruction optional -- if use it, need monitored stack
    (perform (op print-stack-statistics))
@@ -147,6 +168,32 @@
    (perform (op user-print) (reg val))
    (goto (label read-eval-print-loop))

+;; caller must set up exp and env. does a normal eval, but then
+;; forces the result if it's a thunk
+ev-actual-value
+   (save continue)
+   (save exp)
+   (save env)
+   (assign continue (label ev-force-it))
+   (goto (label eval-dispatch))
+ev-force-it
+   (test (op thunk?) (reg val))
+   (branch (label ev-force-thunk))
+   (goto (label ev-actual-value-done))
+ev-force-thunk
+   (assign exp (op thunk-exp) (reg val))
+   (assign env (op thunk-env) (reg val))
+   (assign continue (label ev-actual-value-done))
+   (goto (label ev-actual-value))
+ev-actual-value-done
+   (restore env)
+   (restore exp)
+   (restore continue)
+   (goto (reg continue))
+ev-delay-it
+   (assign val (op delay-it) (reg exp) (reg env))
+   (goto (reg continue))
+
; ;SECTION 5.4.1
eval-dispatch
    (test (op self-evaluating?) (reg exp))
@@ -196,7 +243,7 @@
    (save unev)
    (assign exp (op operator) (reg exp))
    (assign continue (label ev-appl-did-operator))
-   (goto (label eval-dispatch))
+   (goto (label ev-actual-value))
ev-appl-did-operator
    (restore unev)
    (restore env)
@@ -205,6 +252,16 @@
    (test (op no-operands?) (reg unev))
    (branch (label apply-dispatch))
    (save proc)
+ev-maybe-delay
+   ; ; we need to either force or delay the arguments. rather than duplicating instructions,
+   ; ; we will store which one to do in "proc". Proc is saved for us above.
+   (test (op compound-procedure?) (reg proc))
+   (branch (label ev-set-delay))
+   (assign proc (label ev-actual-value))
+   (goto (label ev-appl-operand-loop))
+ev-set-delay
+   (assign proc (label ev-delay-it))
+   (goto (label ev-appl-operand-loop))
ev-appl-operand-loop
    (save argl)
    (assign exp (op first-operand) (reg unev))
@@ -213,8 +270,10 @@
    (save env)
    (save unev)
    (assign continue (label ev-appl-accumulate-arg))
-   (goto (label eval-dispatch))
+   (save proc)
+   (goto (reg proc)) ; ; use the flag we set earlier
ev-appl-accumulate-arg
+   (restore proc)
    (restore unev)
    (restore env)
    (restore argl)
@@ -282,7 +341,7 @@
    (save continue)
    (assign continue (label ev-if-decide))
    (assign exp (op if-predicate) (reg exp))
-   (goto (label eval-dispatch))
+   (goto (label ev-actual-value))

```

```
ev-if-decide
  (restore continue)
  (restore env)
@@ -371,17 +430,16 @@

  (define the-global-environment (setup-environment))
 ;(eceval 'trace-on)
-;((eceval 'set-breakpoint) 'ev-cond-do 4)
-; (set-register-trace! eceval 'exp #t)
-; (set-register-trace! eceval 'unev #t)
+;((eceval 'set-breakpoint) 'ev-actual-value-done 3)
+;(set-register-trace! eceval 'exp #t)
+
  (start eceval)

-(define (f a b)
-  (let ((x a) (y b))
-    (cond ((> x y) 'first)
-          ((< x y) 'second)
-          (else 'neither))))
-
- (f 5 7)
- (f 7 5)
- (f 5 5)
\ No newline at end of file
+(define (unless test usual exceptional)
+  (if test exceptional usual))
+
+(define (divide-unless a b)
+  (unless (= b 0) (/ a b) 'bullshit))
+
+(divide-unless 15 5)
+(divide-unless 3 0)
```

thanhnghuyen2187

I wrote the solution with detailed explanation here:  
<https://nguyễnhuythanh.com/posts/sicp-5.25/>

# sicp-ex-5.26

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (5.25) | Index | Next exercise (5.27) >>

meteorgan

(a)  
n      total-pushes      maximum-depth  
1      64                    10  
2      99                    10  
3      134                  10  
so the maximum-depth is 10  
(b)  
total-pushes =  $35 \times n + 29$

codybartfast

By the way, the maximum depth is not constant with Normal application:

	Maximum Depth	Number of Pushes
Factorial Applicative	$0n + 10$	$35n + 35$
Factorial Normal	$6n + 3$	$50n + 48$

The stack does not grow during the construction of the answer, instead the answer contains nested thunks. So I think forcing the answer to an actual value causes recursive calls to actual-value, eval-dispatch, and force-it, which grows the stack. Of course a better design than mine might eliminate this, especially if we didn't have memoization.

Last modified : 2020-02-06 09:46:25  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-5.27

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (5.26) | Index | Next exercise (5.28) >>

meteorgan		
	maximum-depth	total-pushes
recursion	$5n+3$	$32n-16$
iteration	10	$35n+29$

donald		
	maximum-depth	total-pushes
recursion	$5n+3$	$32n-16$
iteration	10	$35n+34$

Last modified : 2015-12-01 16:34:12  
WiLiKi 0.5-tekili-7 running on Gauche 0.9

# sicp-ex-5.28

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (5.27) | Index | Next exercise (5.29) >>

meteorgan

	total-pushes	maximum-depth
recursion	$37n + 33$	$3n + 14$
iteration	$34n - 16$	$8n + 3$

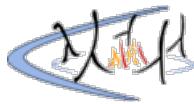
-----  
Either this one is backwards **or** the one above.

	Maximum depth	Number of pushes
Recursive Factorial	$8n+3$	$34n-16$
Iterative Factorial	$3n+14$	$37n+33$
Recursive Factorial		

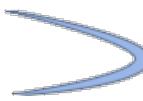
codybartfast

Reproduced the same results (except for constants):

	Maximum Depth	Number of Pushes
Recursive Factorial	$34n + -8$	$8n + 6$
Iterative Factorial	$37n + 41$	$3n + 17$



# sicp-ex-5.29



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.28) | Index | Next exercise (5.30) >>

meteorgan

(a)  
n total-pushes maximum-pushes  
1 18 11  
2 78 19  
3 138 27  
4 258 35  
5 438 43  
the maximum-pushes is:  $8n + 3$   
(b)  
 $S(n) = S(n-1) + S(n-2) + 42$   
 $S(n) = 60Fib(n+1) - 42$

codybartfast

Results

n	Fibonacci	Maximum Depth	Number of Pushes
1	1	8	22
2	1	13	78
3	2	18	134
4	3	23	246
5	5	28	414
6	8	33	694
7	13	38	1,142
8	21	43	1,870
9	34	48	3,046
10	55	53	4,950
11	89	58	8,030

Part A

=====

$$\text{Max pushes} = 5n + 3$$

Part B

=====

Formula for Number of Pushes wrt previous values:

$$S(n) = S(n-1) + S(n-2) + 34$$

Formula for Number of Pushes wrt next Fibonacci number:

$$S(n) = 56 \text{ Fib}(n+1) + -34$$

Calculation:

=====

```

S(2): 78 = a.2 + b
S(3): 134 = a.3 + b

S(3) - S(2):
  (134 - 78) = (3a + b) - (2b + b)
    56 = (3a - 2a) + (b - b)
      a = 56

S(2): 78 = a.2 + b
      78 = (56 * 2) + b
    78 - 112 = b
      b = -34

```

Without Tail Recursion:  
=====

Without tail recursion I got results more closely matching meteorgan's results:

```

Max pushes = 8n + 6
S(n) = 60 Fib(n+1) + -34

```

anon

I find it very strange that we all have different a, b and K for fib, the max-depth formula is the same as cody's, while  
 $S(n) = \text{fib}(n+1)*(S(1) + K) - K;$   
so  $a = S(1) + K$   
 $b = -K$   
also  $S(1) = S(0)$   
but my values differ from others  
 $S(1) = 16$   
 $K = 40$   
 $S(n) = 56*\text{fib}(n+1) - 40$   
Also the order of growth is  $\phi^{n+1}$  as  $\text{fib}(n+1) = (\phi^{n+1} - (1 - \phi)^{n+1})/\sqrt{5}$   
which is equivalent to  $\phi^n$

ama

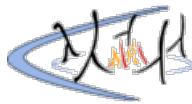
```

;; a)
;;
;; Fibonacci: Space: s(n)=3+8n, n>0; s(0)=11.
;; n = 0      pushes = 18      depth = 11      v = 0
;; n = 1      pushes = 18      depth = 11      v = 1
;; n = 2      pushes = 78      depth = 19      v = 2
;; n = 3      pushes = 138     depth = 27      v = 3
;; n = 4      pushes = 258     depth = 35      v = 4
;; n = 5      pushes = 438     depth = 43      v = 5
;; n = 6      pushes = 738     depth = 51      v = 6
;; n = 7      pushes = 1218    depth = 59      v = 7
;; n = 8      pushes = 1998    depth = 67      v = 8
;; n = 9      pushes = 3258    depth = 75      v = 9
;; n = 10     pushes = 5298    depth = 83      v = 10

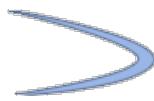
;; b)
;;
;; S(n)=42+S(n-1)+S(n-2)
;;
;; s(n) = k + s(n-1) + s(n-2)
;;
;; a = 18; k = 42
;;
;; s(0) =      a          = 18
;; s(1) =      a          = 18
;; s(2) =      2a + k    = 78
;; s(3) =      3a + 2k   = 138
;; s(4) =      5a + 4k   = 258
;; s(5) =      8a + 7k   = 438
;; s(6) =      13a + 12k = 738
;;
;; o
;;
;; o
;;
;; o
;;
;; s(n) = Fib(n+1) * a + (Fib(n+1) - 1) * k
;;

```

```
;; s(n) = 60 Fib(n+1) - 42
```



# sicp-ex-5.30



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.29) | Index | Next exercise (5.31) >>

meteorgan

```
; if the variable is unbounded, cons 'unbound with it;
; otherwise, cons 'bounded with it. then we can use
; car to judge it's bounded or not.
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
              (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (cons 'bounded (car vals))) ; ****
            (else (scan (cdr vars) (cdr vals))))))
      (if (eq? env the-empty-environment)
          (cons 'unbounded '())
          (let ((frame (first-frame env)))
            (scan (frame-variables frame)
                  (frame-values frame)))))
    (env-loop env))
  (define (bound-variable? var)
    (and (pair? var) (eq? (car var) 'bounded)))
  (define (extract-variable-value var)
    (cdr var))

ev-variable
  (assign val (op lookup-variable-value) (reg exp) (reg env))
  (test (op bound-variable?) (reg val))
  (branch (label bound-variable))
  (assign val (const unbounded-variable-error))
  (goto (label signal-error))
bound-variable
  (assign val (op extract-variable-value) (reg val))
  (goto (reg continue))
```

Rptx

The same thing must also be done with ev-assignment. And set-variable-value. And for b.

```
(define safe-primitives
  (list car cdr /))
(define (apply-primitive-procedure proc args)
  (if debug
      (display (list 'apply-primitive proc args)) (newline))
  (let ((primitive (primitive-implementation proc)))
    (if (member primitive safe-primitives)
        (safe-apply primitive args) ; ex 5.30 b
        (apply-in-underlying-scheme
         primitive args)))

(define (safe-apply proc args) ; ex 5.30 b
  (if debug
      (display 'safe-apply) (newline))
  (cond ((or (eq? proc car)
             (eq? proc cdr))
         (safe-car-cdr proc args))
        ((eq? proc /)
         (safe-division proc args))
        (else
         (list 'primitive-error proc args)))))

(define (primitive-error? val) ; ex 5.30 b
  (tagged-list? val 'primitive-error))

(define (safe-car-cdr proc args) ; ex 5.30 b
  (if debug
      (display (list 'safe-car-cdr args))
      (newline))
  (if (not (pair? (car args))) ; args is a list (args '())
      (error "bad arguments to CAR"))
  (list (safe-car proc args) (safe-cdr proc args))))
```

```

(list 'primitive-error 'arg-not-pair)
      (apply-in-underlying-scheme proc args)))
(define (safe-division proc args)           ; ex 5.30 b
  (if (= 0 (cadr args))
      (cons 'primitive-error 'division-by-zero)
      (apply-in-underlying-scheme proc args)))

; and this is added to the evaluator.

primitive-apply
  (assign val (op apply-primitive-procedure)
         (reg proc)
         (reg argl))
  (test (op primitive-error?) (reg val)) ; ex 5.30 b
  (branch (label primitive-error))
  (restore continue)
  (goto (reg continue))

primitive-error
  (restore continue)                   ; clean up stack from apply-dispatch
  (goto (label signal-error))

```

revc

NOTE: I use chez scheme as the implementation of Scheme.

```

;; some supporting procedures or objects

;; (gensym) procedure
;; returns: a unique generated symbol
;; libraries: (chezscheme)

(define *UNBOUNDED-ERROR* (gensym))
(define (unbounded-error? err) (eq? err *UNBOUNDED-ERROR*))

(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
              (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        *UNBOUNDED-ERROR*
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame))))))
  (env-loop env))

(define *NOT-PAIR-ERROR* (gensym))
(define (not-pair-error? err) (eq? err *NOT-PAIR-ERROR*))

(define *INCORRECT-ARITY-ERROR* (gensym))
(define (incorrect-arity-error? err) (eq? err *INCORRECT-ARITY-ERROR*))

(define *NOT-NUMBER-ERROR* (gensym))
(define (not-number-error? err) (eq? err *NOT-NUMBER-ERROR*))

(define *DIVISION-ZERO-ERROR* (gensym))
(define (division-zero-error? err) (eq? err *DIVISION-ZERO-ERROR*))

;; (procedure-arity-mask proc) procedure
;; returns: an exact integer bitmask identifying the accepted argument counts of proc
;; libraries: (chezscheme)

;; The bitmask is represented as two's complement number with the bit at each index n set
;; if and only if proc accepts n arguments.
;; The two's complement encoding implies that if proc accepts n or more arguments, the
;; encoding is a negative number, since all the bits from n and up are set. For example,
if
;; proc accepts any number of arguments, the two's complement encoding of all bits set is
-1.

;; (logbit? index int) procedure
;; returns: #t if the specified bit is set, otherwise #f
;; libraries: (chezscheme)

(define (correct-arity? impl args)
  (let ([arity-mask (procedure-arity-mask impl)])
    (logbit? (length args) arity-mask)))

```

```

(define THE-ARITHMETIC-OPERATIONS (list + - * = / > <))
(define (arithmetic-operation? op) (memq op THE-ARITHMETIC-OPERATIONS))
(define (apply-primitive-procedure proc args)
  (let ([impl (primitive-implementation proc)])
    (cond [(not (correct-arity? impl args)) *INCORRECT-ARITY-ERROR*]
          [|(or (eq? impl car) (eq? impl cdr)) (apply-safe-car&cdr impl args)|]
          [(arithmetic-operation? impl) (apply-safe-arithmetic impl args)]
          [else (apply-in-underlying-scheme impl args)])))

(define (all-number-arguments? args)
  (andmap number? args))

(define (apply-safe-arithmetic impl args)
  (cond [(not (all-number-arguments? args)) *NOT-NUMBER-ERROR*]
        [|(or (eq? impl /)) (apply-safe-division args)|]
        [else (apply-in-underlying-scheme impl args)]))

(define (apply-safe-car&cdr impl args)
  (cond [(not (pair? (car args))) *NOT-PAIR-ERROR*]
        [else (apply-in-underlying-scheme impl args)]))

(define (apply-safe-division args)
  (if (zero? (car (last-pair args)))
      *DIVISION-ZERO-ERROR*
      (apply-in-underlying-scheme / args)))

(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (not (= (length vars) (length vals)))
          *INCORRECT-ARITY-ERROR*)))

;;; some controller code fragments
unbounded-val
(assign val (const unbounded-value-error))
(goto (label signal-error))

incorrect-arity
(assign val (const incorrect-arity-error))
(goto (label signal-error))

division-zero
(assign val (const division-zero-error))
(goto (label signal-error))

not-pair
(assign val (const not-pair-error))
(goto (label signal-error))

not-number
(assign val (const not-number-error))
(goto (label signal-error))

ev-variable
(assign val (op lookup-variable-value) (reg exp) (reg env))
(test (op unbounded-error?) (reg val))
(branch (label unbounded-val))
(goto (reg continue))

primitive-apply
(assign val (op apply-primitive-procedure)
       (reg proc)
       (reg argl))
(restore continue)

(test (op incorrect-arity-error?) (reg val))
(branch (label incorrect-arity))

(test (op division-zero-error?) (reg val))
(branch (label division-zero))

(test (op not-pair-error?) (reg val))
(branch (label not-pair))

(test (op not-number-error?) (reg val))
(branch (label not-number))
(goto (reg continue))

compound-apply
(assign unev (op procedure-parameters) (reg proc))
(assign env (op procedure-environment) (reg proc))
(assign env (op extend-environment)
       (reg unev) (reg argl) (reg env))

(test (op incorrect-arity-error?) (reg env))
(branch (label incorrect-arity))

```

```

(assign unev (op procedure-body) (reg proc))
(goto (label ev-sequence))

;;;;;;;;
;;;;;TEST;;;;;;
;;;;;;;;
;;; EC-Eval input:
(define (square x) (* x x))

(total-pushes = 3 maximum-depth = 3)
;;; EC-Eval value:
ok

;;; EC-Eval input:
(square 2 2)
incorrect-arity-error

;;; EC-Eval input:
(square 'x)
not-number-error

;;; EC-Eval input:
(square a)
unbounded-value-error

;;; EC-Eval input:
(+)

(total-pushes = 3 maximum-depth = 3)
;;; EC-Eval value:
0

;;; EC-Eval input:
(-)
incorrect-arity-error

;;; EC-Eval input:
(car 2 2)
incorrect-arity-error

;;; EC-Eval input:
(car (cons 1 2))

(total-pushes = 13 maximum-depth = 8)
;;; EC-Eval value:
1

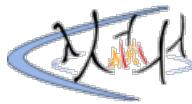
;;; EC-Eval input:
(car 1)
not-pair-error

;;; EC-Eval input:
(/ 1 1 1 1 0)
division-zero-error

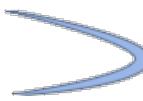
;;; EC-Eval input:
(/ 1 0)
division-zero-error

;;; EC-Eval input:
(/ 0)
division-zero-error

```



# sicp-ex-5.31



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.30) | Index | Next exercise (5.32) >>

meteorgan

(f 'x 'y) => all the saves **and** restores are superfluous. because they doesn't change the registers.  
((f) 'x 'y) => all the saves **and** restores are superfluous.  
(f (g 'x) y) => register proc, argl will needed save **and** restore.  
(f (g 'x) 'y) => same to the above one.

Perry

(f 'x 'y) => all the saves are **not** necessary.  
((f) 'x 'y) => **not** necessary too, although (f) is a compound-procedure, it will change the env register to the status where f is defined, but it will **not** affect the quoted 'x **and** 'y.  
(f (g 'x) y) => proc **and** argl are necessary, save env before **eval** (g 'x) is necessary too, it will change the env register **and** y will be affected.  
(f (g 'x) 'y) => proc **and** argl is necessary, but no need to save env before **eval** (g 'x) cause 'y is quoted, it will **not** fetch any value in the env, **and** env register will be reset to the status where f is defined

codybartfast

I think this matches Perry's answer

	operator env	operands env	operands argl	op'd seq proc
1: (f 'x 'y)	eliminate	eliminate	eliminate	eliminate
2: ((f) 'x 'y)	eliminate	eliminate	eliminate	eliminate
3: (f (g 'x) y)	eliminate			
4: (f (g 'x) 'y)	eliminate	eliminate		

Operator Env  
=====

Only need to save/restore if env could change in evaluating the operator and we need env to evaluate the operands. In the case of 2: the environment would be changed in evaluating (f) but the environment is not needed to evaluate symbols 'x and 'y.

Operands Env  
=====

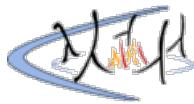
Only need to save/restore between operands if there's an operand expression that might change env and a subsequent operand that needs env for variable lookup. This is only the case for 3:.

Operands Argl  
=====

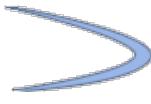
We need to save/restore argl anytime there's an operand that might change the value of argl. That's 3: and 4:.

Operaand Sequence Proc  
=====

We need to save/restore proc if there's any operand that might change the value of proc. That's 3: and 4:.



# sicp-ex-5.32



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.31) | Index | Next exercise (5.33) >>

meteorgan

```
(a)
ev-application
  (save continue)
  (assign unev (op operands) (reg exp))
  (assign exp (op operator) (reg exp))
  (test (op symbol?) (reg exp))           ;; is the operator is symbol?
  (branch (label ev-appl-operator-symbol))
  (save env)
  (save unev)
  (assign continue (label ev-appl-did-operator))
  (goto (label eval-dispatch))
ev-appl-operator-symbol
  (assign continue (label ev-appl-did-operator-no-restore))
  (goto (label eval-dispatch))
ev-appl-did-operator
  (restore unev)
  (restore env)
ev-appl-did-operator-no-restore
  (assign argl (op empty-arglist))
  (assign proc (reg val))      ; the operator
  (test (op no-operands?) (reg unev))
  (branch (label apply-dispatch))
  (save proc)
(b)
It won't get all the advantage of compiler, because the interpreter need parse the code
every time when it run. and recognizing the special case will make the code more
complicated.
```

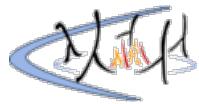
codybartfast

We can use variable? to identify the optimization and then go directly to ev-variable.

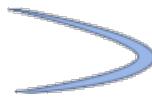
```
ev-application
  (save continue)
  (assign unev (op operands) (reg exp))
  (assign exp (op operator) (reg exp))
  (test (op variable?) (reg exp))        ; check if a variable
  (branch (label ev-appl-operator-lookup)) ; --> to lookup
  (save env)                            ; do need eval
  (save unev)                          ; so do need to save
  (assign continue (label ev-appl-did-operator-eval))
  (goto (label eval-dispatch))

ev-appl-operator-lookup                  ; peform lookup
  (assign continue (label ev-appl-did-operator-lookup))
  (goto (label ev-variable))

ev-appl-did-operator-eval              ; return here if we eval'ed
  (restore unev)
  (restore env)
ev-appl-did-operator-lookup            ; return here if we looked up
  (assign argl (op empty-arglist))
  (assign proc (reg val))
  ...
```



# sicp-ex-5.33



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

---

[<< Previous exercise \(5.32\)](#) | [Index](#) | [Next exercise \(5.34\) >>](#)

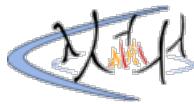
---

meteorgan

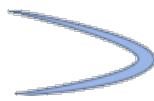
In (\* (factorial (- n 1) n)), compiler need (save argl), then (restore argl).  
In (\* n (factorial (- n 1))), compiler need (save env), then (restore env).  
so there is no efficiency between the two program.

---

Last modified : 2013-06-23 16:35:20  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-5.34



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.33) | Index | Next exercise (5.35) >>

meteorgan

The difference is in (\* n (factorial (- n 1))), before call (factorial (- n 1)), compiler need save continue, then restore it. But in iter, just (goto (reg val)), there is no need to callback.

donald

```
(assign val (op make-compiled-procedure) (label entry24) (reg env))
(goto (label after-lambda23))

entry24
(assign env (op compiled-procedure-env) (reg proc))
(assign env (op extend-environment) (const (n)) (reg argl) (reg env))
;;生成(define (iter product counter) ...)
(assign val (op make-compiled-procedure) (label entry29) (reg env))
(goto (label after-lambda28))

entry29
;;设置环境啦
(assign env (op compiled-procedure-env) (reg proc))
(assign env (op extend-environment) (const (product counter)) (reg argl) (reg env))
;;计算(> counter n)
(save continue)
(save env)
(assign proc (op lookup-variable-value) (const >) (reg env))
(assign val (op lookup-variable-value) (const n) (reg env))
(assign argl (op list) (reg val))
(assign val (op lookup-variable-value) (const counter) (reg env))
(assign argl (op cons) (reg val) (reg argl))
(test (op primitive-procedure?) (reg proc))
(branch (label primitive-branch44))

compiled-branch43
(assign continue (label after-call42))
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))

primitive-branch44
(assign val (op apply-primitive-procedure) (reg proc) (reg argl))

after-call42
;;虽然(> counter n)并未改变env。但过程应用的modified是all-reg，下面紧接着是另一个过程应用，need env，所以会产生这个步骤
	restore env)
	restore continue)
(test (op false?) (reg val))
(branch (label false-branch31))

true-branch32
(assign val (op lookup-variable-value) (const product) (reg env))
(goto (reg continue))

false-branch31
(assign proc (op lookup-variable-value) (const iter) (reg env))
;;计算(+ counter 1)，不会马上给回counter，不用担心product
(save continue)
(save proc)
(save env)
(assign proc (op lookup-variable-value) (const +) (reg env))
(assign val (const 1))
(assign argl (op list) (reg val))
(assign val (op lookup-variable-value) (const counter) (reg env))
(assign argl (op cons) (reg val) (reg argl))
(test (op primitive-procedure?) (reg proc))
(branch (label primitive-branch38))
```

```

compiled-branch37
(assign continue (label after-call36))
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))

primitive-branch38
(assign val (op apply-primitive-procedure) (reg proc) (reg argl))

after-call36
(assign argl (op list) (reg val))
;;计算(* product counter)
	restore env)
(save argl)
(assign proc (op lookup-variable-value) (const *) (reg env))
(assign val (op lookup-variable-value) (const counter) (reg env))
(assign argl (op list) (reg val))
(assign val (op lookup-variable-value) (const product) (reg env))
(assign argl (op cons) (reg val) (reg argl))
(test (op primitive-procedure?) (reg proc))
(branch (label primitive-branch35))

compiled-branch34
(assign continue (label after-call33))
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))

primitive-branch35
(assign val (op apply-primitive-procedure) (reg proc) (reg argl))

after-call33
(restore argl)
;;给iter新的argl构建完毕
(assign argl (op cons) (reg val) (reg argl))
(restore proc)
(restore continue)
;;至此，栈又恢复到上次调用iter的情况
(test (op primitive-procedure?) (reg proc))
(branch (label primitive-branch41))

compiled-branch40
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))

primitive-branch41
(assign val (op apply-primitive-procedure) (reg proc) (reg argl))
(goto (reg continue))

;;用不上了
after-call39
after-if30

;;得到(define (iter ..) ...)
after-lambda28
(perform (op define-variable!) (const iter) (reg val) (reg env))
(assign val (const ok))
;;计算(iter 1 1)
(assign proc (op lookup-variable-value) (const iter) (reg env))
;;构建argl
(assign val (const 1))
(assign argl (op list) (reg val))
(assign val (const 1))
(assign argl (op cons) (reg val) (reg argl))
(test (op primitive-procedure?) (reg proc))
(branch (label primitive-branch27))

;;最后一条表达式，所以不用回到after-call25
compiled-branch26
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))

primitive-branch27
(assign val (op apply-primitive-procedure) (reg proc) (reg argl))
(goto (reg continue))

after-call25

after-lambda23
(perform (op define-variable!) (const factorial) (reg val) (reg env))
(assign val (const ok))

```

# sicp-ex-5.35

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

---

<< [Previous exercise \(5.34\)](#) | [Index](#) | [Next exercise \(5.36\)](#) >>

---

meteorgan

```
;; following the code run.  
(define (f x) (+ x (g (+ x 2))))
```

Last modified : 2012-08-15 04:11:26

WiLiKi 0.5-tekili-7 running on **Gauche 0.9**

# sicp-ex-5.36

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (5.35) | Index | Next exercise (5.37) >>

meteorgan

right to left, this is because in construct-arglist, compiler first **reverse** operand-code, then evaluate the parameters from left to right.

To change the order, we can first evaluate the operands from left to right to get the arglist from right to left, then **reverse** the arglist.

```
(define (construct-arglist operand-codes)
  (let ((operand-codes operand-codes))
    (if (null? operand-codes)
        (make-instruction-sequence '() '(argl)
                                  `((assign argl (const ())))
        (let ((code-to-get-last-arg
              (append-instruction-sequences
                (car operand-codes)
                (make-instruction-sequence '(val) '(argl)
                                          `((assign argl (op list) (reg val)))))))
          (if (null? (cdr operand-codes))
              code-to-get-last-arg
              (tack-on-instruction-sequence
                (preserving '(env)
                  code-to-get-last-arg
                  (code-to-get-rest-args
                    (cdr operand-codes)))
                (make-instruction-sequence '() '()
                  `((assign argl (op reverse) (reg argl)))))))))))
```

donald

```
;no need to reverse
(define (construct-arglist operand-codes)
  (if (null? operand-codes)
      (make-instruction-sequence '()
                                '(argl)
                                `((assign argl (const ()))))
      (let ((code-to-get-last-arg
            (append-instruction-sequences
              (car operand-codes)
              (make-instruction-sequence '(val)
                                        '(argl)
                                        `((assign argl (op list) (reg val)))))))
        (if (null? (cdr operand-codes))
            code-to-get-last-arg
            (preserving '(env)
              code-to-get-last-arg
              (code-to-get-rest-args (cdr operand-codes))))))
  (define (code-to-get-rest-args operand-codes)
    (let ((code-for-next-arg
          (preserving '(argl)
            (car operand-codes)
            (make-instruction-sequence
              '(val argl)
              '(argl)
              ;:now use append instead of cons
              `((assign argl (op append) (reg val) (reg argl)))))))
      (if (null? (cdr operand-codes))
          code-for-next-arg
          (preserving '(env)
            code-for-next-arg
            (code-to-get-rest-args (cdr operand-codes)))))))
```

By the way, there is a bug in the above solution.

poly

```
;;now use append instead of cons
'((assign argl (op append) (reg val) (reg argl))))))
.....
```

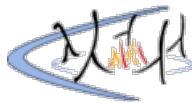
the val should be included in the arglist, which means the code shoule be

```
'((assign val (op list) (reg val))
  (assign argl (op append) (reg argl) (reg val))))))
.....
```

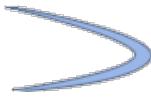
codybartfast

I don't think there is any need to reverse argl.

If we don't reverse operand-codes (as above) we will evaluate arguments left-to-right (instead of right-to-left). This will then result in argl being in 'reverse' order. But we can leave it like this because argl is only used by extend-environment and apply-primitive-procedure and these already expect argl in 'reverse' order because of how the ec-evaluator constructs argl.



# sicp-ex-5.37



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.36) | Index | Next exercise (5.38) >>

meteorgan

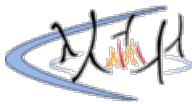
```
; remove the condition, when (preserving regs, seq1 seq2), always (save first-reg),
;; then (restore first-reg)
(define (preserving regs seq1 seq2)
  (if (null? regs)
      (append-instruction-sequences seq1 seq2)
      (let ((first-reg (car regs)))
        (preserving (cdr regs)
                    (make-instruction-sequence
                     (list-union (list first-reg) (registers-needed seq1))
                     (list-difference
                      (registers-modified seq1)
                      (list first-reg))
                     (append
                      `((save ,first-reg)
                        (statements seq1)
                        `((restore ,first-reg))))
                     seq2)))))

;; compare the following code with exercise 5.35
(continue env)
(val)
  (save continue)
  (save env)
  (save continue)
  (assign val (op make-compiled-procedure) (label entry1) (reg env))
  (restore continue)
  (goto (label after-lambda2))
entry1
  (assign env (op compiled-procedure-env) (reg proc))
  (assign env (op extend-environment) (const (x)) (reg argl) (reg env))
  (save continue)
  (save env)
  (save continue)
  (assign proc (op lookup-variable-value) (const +) (reg env))
  (restore continue)
  (restore env)
  (restore continue)
  (save continue)
  (save proc)
  (save env)
  (save continue)
  (save env)
  (save continue)
  (assign proc (op lookup-variable-value) (const g) (reg env))
  (restore continue)
  (restore env)
  (restore continue)
  (save continue)
  (save proc)
  (save continue)
  (save env)
  (save continue)
  (assign proc (op lookup-variable-value) (const +) (reg env))
  (restore continue)
  (restore env)
  (restore continue)
  (save continue)
  (save proc)
  (save env)
  (save continue)
  (assign val (const 2))
  (restore continue)
  (assign argl (op list) (reg val))
  (restore env)
  (save argl)
  (save continue)
  (assign val (op lookup-variable-value) (const x) (reg env))
  (restore continue)
```

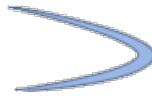
```

  (restore argl)
  (assign argl (op cons) (reg val) (reg argl))
  (restore proc)
  (restore continue)
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch3))
compiled-branch4
  (assign continue (label after-call5))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch3
  (save continue)
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  (restore continue)
after-call5
  (assign argl (op list) (reg val))
  (restore proc)
  (restore continue)
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch6))
compiled-branch7
  (assign continue (label after-call8))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch6
  (save continue)
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  (restore continue)
after-call8
  (assign argl (op list) (reg val))
  (restore env)
  (save argl)
  (save continue)
  (assign val (op lookup-variable-value) (const x) (reg env))
  (restore continue)
  (restore argl)
  (assign argl (op cons) (reg val) (reg argl))
  (restore proc)
  (restore continue)
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch9))
compiled-branch10
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch9
  (save continue)
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  (restore continue)
  (goto (reg continue))
after-call11
after-lambda2
  (restore env)
  (perform (op define-variable!) (const f) (reg val) (reg env))
  (assign val (const ok))
  (restore continue)

```



# sicp-ex-5.38



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.37) | Index | Next exercise (5.39) >>

meteorgan

```
(a)
;; in compile
((open-code? exp) (compile-open-code exp target linkage))

(define (open-code? exp)
  (memq (car exp) '(+ - * /)))

(define (spread-arguments operand1 operand2)
  (let ((op1 (compile operand1 'arg1 'next))
        (op2 (compile operand2 'arg2 'next)))
    (list op1 op2)))

(b)

;; This procedure has a bug. It does not save the environment
;; Around the compilation of the first arg. Because of this it
;; will give incorrect results for recursive procedures. In my answer
;; Below I have fixed this.
(define (compile-open-code exp target linkage)
  (let ((op (car exp))
        (args (spread-arguments (cadr exp) (caddr exp))))
    (end-with-linkage linkage
      (append-instruction-sequences
        (car args)
        (preserving '(arg1))
        (cadr args)
        (make-instruction-sequence '(arg1 arg2) (list target)
          `((assign ,target (op ,op) (reg arg1) (reg arg2))))))))
```

Rptx

```
; (d)
; these include the answer to 5.44

(define (compile-+ exp target linkage compile-time-environment)
  (if (overwrite? (operator exp)
                  compile-time-environment)
      (compile-application exp target linkage compile-time-environment)
      (let ((operands (operands exp)))
        (if (< 2 (length operands))
            (compile (two-by-two '+ operands) target linkage compile-time-environment)
            (let ((operands (spread-arguments operands compile-time-environment)))
              (end-with-linkage
                linkage
                (preserving
                  '(env continue)
                  operands
                  (make-instruction-sequence
                    '(arg1 arg2) (list target)
                    `((assign ,target (op +) (reg arg1) (reg arg2))))))))))

(define (compile-* exp target linkage compile-time-environment)
  (if (overwrite? (operator exp)
                  compile-time-environment)
      (compile-application exp target linkage compile-time-environment)
      (let ((operands (operands exp)))
        (if (< 2 (length operands))
            (compile
              (two-by-two '* operands) target linkage compile-time-environment)
            (let ((operands (spread-arguments operands compile-time-environment)))
              (end-with-linkage
                linkage
                (preserving
                  '(env continue)
                  operands
```

```

(make-instruction-sequence
  '(arg1 arg2) (list target)
  `((assign ,target (op *) (reg arg1) (reg arg2))))))))))

(define (-? exp)
  (tagged-list? exp '-))
(define (compile-- exp target linkage compile-time-environment)
  (let ((operands (spread-arguments (operands exp) compile-time-environment)))
    (end-with-linkage
     linkage
     (preserving
      '(env continue)
      operands
      (make-instruction-sequence
       '(arg1 arg2) (list target)
       `((assign ,target (op -) (reg arg1) (reg arg2)))))))

(define (/? exp)
  (tagged-list? exp '/))
(define (compile-/ exp target linkage compile-time-environment)
  (let ((operands (spread-arguments (operands exp) compile-time-environment)))
    (end-with-linkage
     linkage
     (preserving
      '(env continue)
      operands
      (make-instruction-sequence
       '(arg1 arg2) (list target)
       `((assign ,target (op /) (reg arg1) (reg arg2)))))))

(define (two-by-two proc operands)
  (if (> 2 (length operands))
      (car operands)
      (list proc (car operands)
            (two-by-two proc (cdr operands)))))


```

donald

```

(define (+? exp)
  (tagged-list? exp '+))
;;设定只处理两个参数的情况
(define (spread-arguments arg1)
  (let ((operand-code1 (compile (car arg1) 'arg1 'next))
        (operand-code2 (compile (cadr arg1) 'arg2 'next)))
    (preserving '(env)
      operand-code1
      (make-instruction-sequence
       (list-union '(arg1)
                  (registers-needed operand-code2))
       (list-difference (registers-modified operand-code2)
                       '(arg1))
       (append '((save arg1)
                 (statements operand-code2)
                 '((restore arg1))))))
    (define (compile++ exp target linkage)
      (let ((operand-codes (spread-arguments (operands exp))))
        (end-with-linkage
         linkage
         (preserving '(continue)
           operand-codes
           (make-instruction-sequence
            '()
            `'(target)
            `((assign ,target (op +) (reg arg1) (reg arg2)))))))

(define t7 (compile-test '(+ (+ a 1) (+ 3 2))))
;;the result of t7
(assign arg1 (op lookup-variable-value) (const a) (reg env))
(save arg1)
(assign arg2 (const 1))
	restore arg1)
(assign arg1 (op +) (reg arg1) (reg arg2))
(save arg1)
(assign arg1 (const 3))
(save arg1)
(assign arg2 (const 2))
(restore arg1)
(assign arg2 (op +) (reg arg1) (reg arg2))
(restore arg1)
(assign val (op +) (reg arg1) (reg arg2))

;;d

```

```

(define (compile-++ exp target linkage)
  (compile-+ (construct exp) target linkage))
(define (construct exp)
  (if (> (length (operands exp))
           2)
      (append (list (car exp)
                    (cadr exp))
              (list (append (list (car exp))
                            (caddr exp)))))

      exp))

```

poly

Obviously, there are many redundant work will be created in donald's answer :-)

revc

A solution with verification support (You can quickly verify your answer).

```

(define all-regs '(env proc val arg1 continue arg1 arg2))

;; a clause for the dispatch of compile that handles the application of an open-coded
primitive

((open-code-application? exp) (compile-open-code exp target linkage))

(define (open-code-application? exp)
  (memq (car exp) '(+ - * / =)))

(define (second-operand operands) (cadr operands))

(define (spread-arguments operands)
  (let ([arg1-code (compile (first-operand operands) 'arg1 'next)]
        [arg2-code (compile (second-operand operands) 'arg2 'next)])
    (if (= (length operands) 2)
        (values arg1-code arg2-code)
        (error "Unsupported arity!" operands)))))

(define (compile-open-code exp target linkage)
  (let-values ([(arg1-code arg2-code) (spread-arguments (operands exp))])
    (end-with-linkage
     linkage
     (preserving
      '(env)
      arg1-code
      (preserving
       '(arg1)
       arg2-code
       (make-instruction-sequence
        '(arg1 arg2) ;in fact, arg2 can be omitted.
        (list target)
        `((assign ,target (op ,(operator exp)) (reg arg1) (reg arg2))))))))
    )))
  )

;; verification support
(load "ch5-regsim.scm")
;; modified version
(define (lookup-prim symbol operations) (eval symbol))
;; (define exp '(begin (define (square x) (* x x)) (define x 4) (+ x (square 2))))
;; (define exp '(begin (define (square x) (* x x)) (define x 4) (+ (square 2) x)))

(define exp '(begin
              (define (factorial n)
                (if (= n 1)
                    1
                    (* (factorial (- n 1)) n)))
              (factorial 5)))

(define demo-machine
  (make-machine
   all-regs
   '()
   (statements (compile exp 'val 'next)))))

(define the-global-environment (setup-environment))
(set-register-contents! demo-machine 'env (get-global-environment))
(start demo-machine)
(pretty-print (get-register-contents demo-machine 'val))

;; part d
(define (spread-arguments operands)
  (let ([arg1-code (compile (first-operand operands) 'arg1 'next)])
    (rest-codes (map (lambda (op) (compile op 'arg2 'next))
                     (rest-operands operands)))))


```

```

(if (>= (length operands) 2)
  (cons arg1-code rest-codes)
  (error "Unsupported arity!" operands)))))

(define (compile-open-code exp target linkage)
  (define (compile-open-code-rest operand-codes)
    (if (null? (cdr operand-codes))
        (preserving
         '(arg1)
         (car operand-codes)
         (make-instruction-sequence
          '(arg1 arg2) ;in fact, arg2 can be omitted.
          (list target)
          `((assign ,target (op ,(operator exp)) (reg arg1) (reg arg2)))))

        (preserving
         '(arg1 env)
         (car operand-codes)
         (append-instruction-sequences
          (make-instruction-sequence
           '(arg1 arg2) ;in fact, arg2 can be omitted.
           '(arg1)
           `((assign arg1 (op ,(operator exp)) (reg arg1) (reg arg2))))
          (compile-open-code-rest (cdr operand-codes)))
        ))))

;; we evaluate the first operand and the second operand,
;; then assign the values to the corresponding registers sequentially.
;; By accumulating arg2 into arg1, we have the accumulation of the first
;; two operands. After that, we evaluate the third and put its value into
;; arg2. As above, we accumulate arg2 into arg1 so that we have the
;; accumulation of the first three oprands. And so on, until we reach
;; the last operand of the operands, this time, we put the accumulation
;; of arg1 and arg2 into the target register.

(let ([operand-codes (spread-arguments (operands exp))])
  (end-with-linkage
   linkage
   (preserving
    '(env)
    (car operand-codes)
    (compile-open-code-rest (cdr operand-codes)))))))

```

codybartfast

```

Part A
=====

(define (spread-arguments operands)
  (if (= 2 (length operands))
      (preserving '(env)
                  (compile (car operands) 'arg1 'next)
                  (preserving '(arg1)
                              (compile (cadr operands) 'arg2 'next)
                              (make-instruction-sequence
                               '(arg1 '() '())))
                  (error \"Spread-arguments expects 2 args -- COMPILE\" operands)))
      (error \"Spread-arguments expects 2 args -- COMPILE\" operands)))

Part B
=====

(define (compile exp target linkage)
  (cond ((self-evaluating? exp)
         ...
         ((=? exp) (compile-= exp target linkage))
         ((>*? exp) (compile-* exp target linkage))
         ...
         (else
          (error \"Unknown expression type -- COMPILE\" exp))))
  (define (=? exp) (tagged-list? exp '=))
  (define (compile-= exp target linkage)
    (compile-2arg-open-code '=' (operands exp) target linkage))

  (define (*? exp) (tagged-list? exp '*))
  (define (compile-* exp target linkage)
    (compile-2arg-open-code '*' (operands exp) target linkage))
  ...
  (define (compile-2arg-open-code operator operands target linkage)

```

```

(end-with-linkage
 linkage
 (append-instruction-sequences
  (spread-arguments operands)
  (make-instruction-sequence
   '(arg1 arg2)
   ` (,target)
   ` ((assign ,target (op ,operator) (reg arg1) (reg arg2)))))))

```

Part C

=====

With these modifications there are half as many instructions in the lambda body than before (58 before, 29 after). We could therefore expect it to run about twice as fast.

Part D

=====

```

(define (*? exp) (tagged-list? exp '*))
(define (compile-* exp target linkage)
  (compile-multi-arg-open-code '*' (operands exp) target linkage '1))

(define (+? exp) (tagged-list? exp '+))
(define (compile+ exp target linkage)
  (compile-multi-arg-open-code '+' (operands exp) target linkage '0))

(define (compile-multi-arg-open operator operands target linkage op-id)
  (let ((operand-count (length operands)))
    (cond
      ((= 0 operand-count) (compile op-id target linkage))
      ((= 1 operand-count) (compile (car operands) target linkage))
      (else
        (end-with-linkage
         linkage
         (preserving
          '(env)
          (compile (car operands) 'arg1 'next)
          (compile-open-code-reduce operator (cdr operands) target)))))))

(define (compile-open-code-reduce operator operands target)
  (let* ((is-last-operand (null? (cdr operands)))
         (trgt (if is-last-operand target 'arg1))
         (open-code-apply
          (preserving 'arg1)
          (compile (car operands) 'arg2 'next)
          (make-instruction-sequence
           '(arg1 arg2)
           ` (,trgt)
           ` ((assign ,trgt (op ,operator)
                     (reg arg1) (reg arg2)))))))
    (if is-last-operand
        open-code-apply
        (preserving
         '(env)
         open-code-apply
         (compile-open-code-reduce operator (cdr operands) target)))))


```

Simple Example

=====

```

(compile
  `(+ 1 2 3 4 5)
  'val
  'next)

```

Output:

-----

```

()
(arg1 arg2 val)
((assign arg1 (const 1))
 (assign arg2 (const 2))
 (assign arg1 (op +) (reg arg1) (reg arg2))
 (assign arg2 (const 3))
 (assign arg1 (op +) (reg arg1) (reg arg2))
 (assign arg2 (const 4))
 (assign arg1 (op +) (reg arg1) (reg arg2))
 (assign arg2 (const 5))
 (assign val (op +) (reg arg1) (reg arg2)))

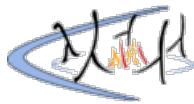
```

More Complex Example

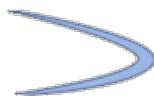
=====

```
(compile
  '(* (* (* 2) (values 3) (* 4 five))
  'val
  'return)

Output:
-----
((env continue)
 (proc arg1 arg1 arg2 val)
 ((save continue)
  (assign arg1 (const 1))
  (assign arg2 (const 2))
  (assign arg1 (op *) (reg arg1) (reg arg2)))
 (save env)
 (assign proc (op lookup-variable-value) (const values) (reg env))
 (assign val (const 3))
 (assign arg1 (op list) (reg val))
 (test (op primitive-procedure?) (reg proc))
 (branch (label primitive-branch1))
 compiled-branch2
 (assign continue (label proc-return4))
 (assign val (op compiled-procedure-entry) (reg proc))
 (goto (reg val))
 proc-return4
 (assign arg2 (reg val))
 (goto (label after-call3))
 primitive-branch1
 (assign arg2 (op apply-primitive-procedure) (reg proc) (reg arg1))
 after-call3
 (assign arg1 (op *) (reg arg1) (reg arg2))
 (restore env)
 (save arg1)
 (assign arg1 (const 4))
 (assign arg2 (op lookup-variable-value) (const five) (reg env))
 (assign arg2 (op *) (reg arg1) (reg arg2))
 (restore arg1)
 (assign val (op *) (reg arg1) (reg arg2))
 (restore continue)
 (goto (reg continue))))
```



# sicp-ex-5.39



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.38) | Index | Next exercise (5.40) >>

meteorgan

```
(define (lexical-address addr-frame addr-offset)
  (cons addr-frame addr-offset))
(define (addr-frame address) (car address))
(define (addr-offset address) (cdr address))

(define (lexical-address-lookup env address)
  (let* ((frame (list-ref env (addr-frame address)))
         (value (list-ref (frame-values frame) (addr-offset address))))
    (if (eq? value '*unassigned*)
        (error "the variable is unassigned -- LEXICAL-ADDRESS-LOOKUP" address)
        value))

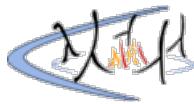
(define (lexical-address-set! env address value)
  (let ((frame (addr-frame address))
        (offset (addr-frame address)))
    (define (set-value! f pos)
      (if (= f 0)
          (set-car! f value)
          (set-value! (cdr f (- pos 1))))))
    (set-value! frame offset value)))
```

donald

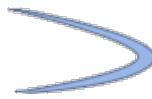
```
;;my frame looks like this
(define (make-frame variables values)
  (map list variables values))

(define (lexical-address-lookup address env)
  (let ((value (cadr (list-ref (list-ref env
                                         (car address))
                               (cadadr address)))))

    (if (eq? value
              '*unassigned*)
        (error "Unassigned variable! -- LEXICAL-ADDRESS-LOOKUP" address)
        value)))
(define (lexical-address-set! address env value)
  (let ((binding (list-ref (list-ref env
                                       (car address))
                           (cadadr address)))
        (set-cdr! binding value)))
```



# sicp-ex-5.40



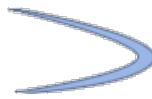
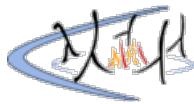
[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.39) | Index | Next exercise (5.41) >>

meteorgan

```
(define (compile-lambda-body exp proc-entry ct-env)
  (let ((formals (lambda-parameters exp)))
    (append-instruction-sequences
      (make-instruction-sequence '(env proc argl) '(env)
        `,(proc-entry
          (assign env (op compiled-procedure-env) (reg proc))
          (assign env
            (op extend-environment)
            (const ,formals)
            (reg argl)
            (reg env))))
      (compile-sequence
        (lambda-body exp)
        'val 'return
        (extend-ct-env ct-env formals))))))
(define (extend-ct-env env frame)
  (cons frame env))
```

Last modified : 2013-06-23 16:33:40  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



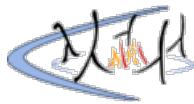
<< Previous exercise (5.40) | Index | Next exercise (5.42) >>

meteorgan

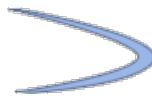
```
(define (find-variable variable lst)
  (define (search-variable v l n)
    (cond ((null? l) false)
          ((eq? v (car l)) n)
          (else (search-variable v (cdr l) (+ n 1)))))
  (define (search-frame frames f)
    (if (null? frames)
        'not-found
        (let ((o (search-variable variable (car frames) 0)))
          (if o
              (cons f o)
              (search-frame (cdr frames) (+ f 1)))))))
  (search-frame lst 0))
```

donald

```
(define (find-variable var ct-env)
  (let ((f-ref 0))
    (define (find var frame)
      (if (eq? var (car frame))
          0
          (+ 1 (find var (cdr frame)))))
    (define (iter ct-env)
      (cond ((null? ct-env)
             'not-found)
            ((memq? var (car ct-env))
             (list f-ref (find var (car ct-env))))
            (else (set! f-ref (+ 1 f-ref))
                  (iter (cdr ct-env))))))
    (iter ct-env)))
```



# sicp-ex-5.42



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.41) | Index | Next exercise (5.43) >>

meteorgan

```
; I skip compile-assignment.  
(define (compile-variable exp target linkage ct-env)  
  (let ((r (find-variable exp ct-env)))  
    (if (eq? r 'not-found)  
        (end-with-linkage linkage  
          (make-instruction-sequence '(env) (list target)  
            `((assign ,target  
                (op lookup-variable-value)  
                (const ,exp)  
                (reg env))))  
        (end-with-linkage linkage  
          (make-instruction-sequence '(env) (list target)  
            `((assign ,target  
                (op lexical-address-lookup)  
                (const ,r)  
                (reg env)))))))
```

Rptx

```
(define (compile-variable exp target linkage compile-time-environment)  
  (let ((lexical-addr (find-variable exp compile-time-environment)))  
    (end-with-linkage  
      linkage  
      (make-instruction-sequence  
        '(env) (list target)  
        (if (eq? 'not-found lexical-addr)  
            `((assign ,target  
                (op lookup-variable-value)  
                (const ,exp)  
                (reg env)))  
            `((assign ,target  
                (op lexical-address-lookup)  
                (const ,lexical-addr)  
                (reg env)))))))  
  
(define (compile-assignment exp target linkage compile-time-environment)  
  (let ((var (assignment-variable exp))  
        (get-value-code  
          (compile (assignment-value exp) 'val 'next compile-time-environment)))  
    (let ((lexical-addr (find-variable var compile-time-environment)))  
      (end-with-linkage  
        linkage  
        (preserving  
          '(env)  
          get-value-code  
          (make-instruction-sequence  
            '(env val) (list target)  
            (if (eq? lexical-addr 'not-found)  
                `((perform (op set-variable-value!)  
                  (const ,var)  
                  (reg val)  
                  (reg env))  
                (assign ,target (const ok)))  
                `((perform (op lexical-address-set!)  
                  (const ,lexical-addr)  
                  (reg val)  
                  (reg env)))))))  
  
    (compile '(((lambda (x y)  
                 (+ x y))  
               1 2)  
              'val  
              'next  
              the-empty-environment))
```

```

; after compiling, and simulating -> 3

(compile '(((lambda (x y)
    (lambda (a b c d e)
        ((lambda (y z) (* x y z))
         (* a b x)
         (+ c d x))))
    3 4)
  1 2 3 4 5)
  'val
  'next
  the-empty-environment)
; generates and 89 instruction sequence machine, and computes 180.
; as expected.

```

aos

Similar to what you all had, but I instead looked in the global environment if I got back `not-found`:

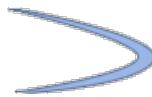
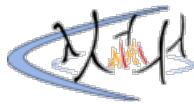
```

(define (compile-variable
      exp target linkage compile-env)
  (let ((var-addr (find-variable exp
                                 compile-env)))
    (end-with-linkage
     linkage
     (make-instruction-sequence
      '(env)
      (list target)
      (if (eq? var-addr 'not-found)
          `((save env)
            (assign env (op get-global-environment))
            (assign ,target
                    (op lookup-variable-value)
                    (const ,exp)
                    ; Look for it straight in global env
                    (reg env)))
          (restore env))
        `((assign ,target
                  (op lexical-address-lookup)
                  (const ,var-addr)
                  (reg env)))))))

```

someone

I think this could be a little better; You might as well assign the global environment to `,target` instead of `env`. That way you can get rid of the `(save env)` and `(restore env)` instructions.



<< Previous exercise (5.42) | Index | Next exercise (5.44) >>

meteorgan

```
; we just need use scan-out-defines here change lambda-body to
; equivalent expression.
(define (compile-lambda-body exp proc-entry ct-env)
  (let ((formals (lambda-parameters exp)))
    (append-instruction-sequences
      (make-instruction-sequence '(env proc argl) '(env)
        `,(proc-entry
          (assign env (op compiled-procedure-env) (reg proc))
          (assign env
            (op extend-environment)
            (const ,formals)
            (reg argl)
            (reg env))))
      (compile-sequence
        (scan-out-defines (lambda-body exp))
        'val 'return
        (extend-ct-env ct-env formals))))
```

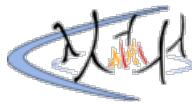
codybartfast

I needed to make two changes to the original scan-out-defines:

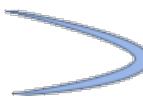
1. Instead of transforming to a let expression (as implied by 4.1.6), I transformed directly to a lambda expression. Alternatively, I could have added let support to the compiler.
2. \*unassigned\* needed to be double quoted, i.e.:

'"unassigned"

otherwise, when the compiler analyzed the lambda expression it would interpret \*unassigned\* as a variable name.



# sicp-ex-5.44



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.43) | Index | Next exercise (5.45) >>

meteorgan

```
(define (overwrite? operator ct-env)
  (let ((r (find-variable operator ct-env)))
    (eq? r 'not-found)))
(define (open-code? exp ct-env)
  (and (memq (car exp) '(+ - * /))
       (overwrite? (car exp) ct-env)))
```

Rptx

My modified procedures are in [Exercise 5.38](#)

codybartfast

Just to expand on the comment in the question that "The code will work correctly as long as the program does not define or set! these names."

This approach will work with:

```
((lambda (+ a b)
         (+ a b))
 - 10 3)
```

which means it will also work with internal definitions:

```
(define (seven)
  (define + -)
  (+ 10 3))
```

But it won't work with a top level define:

```
(define + -)
(+ 10 3)
```

because this definition is stored in the global environment which isn't available at compile time. Nor will it work with:

```
(define (seven)
  (set! + -)
  (+ 10 3))
```

set! will fail because there is no + defined in the environment.

revc

A compilation of solutions from Ex 5.39 to Ex 5.44.

```
(load "ch5-compiler.scm")

;; syntax support
(define the-open-code-procedures
  '(+ - * / =))

(define (open-code-application? exp)
  (memq (car exp) the-open-code-procedures))

(define (second-operand operands) (cadr operands))

;; Exercise 5.39-5.42 implement an "real" definition version of lexical addressing
```

```

compiler,
;;; in which internal definitions for block structure are considered "real"
;;; defines.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; NOTE: YOU CAN COMMENT OUT EXERCISE 5.43 IF YOU WANT TO RUN THIS VERSION.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Exercise 5.39
;;; operations used by run-time environment for lexical-addressing

(define (frame-number address)
  (car address))

(define (displacement-number address)
  (cadr address))

;; return a frame with a given index
(define (env-ref runtime-env n)
  (if (= n 0)
      (first-frame runtime-env)
      (env-ref (enclosing-environment runtime-env)
              (- n 1)))))

(define (frame-values-ref frame-values n)
  (list-ref frame-values n))

(define (frame-values-set! frame-values n val)
  (if (zero? n)
      (set-car! frame-values val)
      (frame-values-set! (cdr frame-values) (- n 1) val)))

(define (lexical-address-lookup address runtime-env)
  (let* ([f-num (frame-number address)]
        [d-num (displacement-number address)]
        [val (frame-values-ref (env-ref runtime-env f-num)
                               d-num)])
    (if (eq? val '*unassigned*)
        (error "use unassigned variable" (car vars))
        val)))

(define (lexical-address-set! address val runtime-env)
  (let ([f-num (frame-number address)]
        [d-num (displacement-number address)])
    (frame-values-set!
     (frame-values
      (env-ref runtime-env f-num))
     d-num
     val)))

;;; Exercise 5.40

;; a blacklist contains a list of modified open-code-procedures
;; which are no longer considered as open-code-procedures,
;; we won't compile modified open-code-procedures "in line",
;; instead, those will be taken as general application.
(define (compile exp compile-time-env blacklist target linkage)
  (cond ((self-evaluating? exp)
         (compile-self-evaluating exp target linkage))
        ((quoted? exp) (compile-quoted exp target linkage))
        ((variable? exp)
         (compile-variable exp compile-time-env target linkage))
        ((assignment? exp)
         (compile-assignment exp compile-time-env blacklist target linkage))
        ((definition? exp)
         (compile-definition exp compile-time-env blacklist target linkage))
        ((if? exp) (compile-if exp compile-time-env blacklist target linkage))
        ((lambda? exp) (compile-lambda exp compile-time-env blacklist target linkage))
        ((begin? exp)
         (compile-sequence (begin-actions exp)
                           compile-time-env
                           blacklist
                           target
                           linkage))
        ((cond? exp) (compile (cond->if exp) compile-time-env blacklist target linkage))
        ;; LET clause for the scaping out version
        ((let? exp) (compile (let->combination exp) compile-time-env blacklist target linkage))
        ((and (open-code-application? exp) ; we compile exp "in line" if and only if
              ; exp is open-code-application [1]
              ;
              ; and operator is open-code-procedure
              ; (which means operator is not in
              ; blacklist) [2]
              ;

```

```

; and variable will be found in top-level-
environment
; if we are in the current compile-time
environment [3]
;
(open-code-procedure? (operator exp) blacklist)
(variable-in-top-level? (operator exp) compile-time-env))
(compile-open-code exp compile-time-env blacklist target linkage))
((application? exp)
(compile-application exp compile-time-env blacklist target linkage))
(else
(error "Unknown expression type -- COMPILE" exp)))

(define (compile-sequence seq compile-time-env blacklist target linkage)
;; for internal definitions without scanning out, we need
;; impose an evaluation order on sequence evaluation so that
;; subsequent expressions can refer to preceding definitions.
;; Here's why: I found the order of evaluation of operands is
;; dependent on the implementation of Scheme. In Chez Scheme,
;; the interpreter does not follow ours intuition that
;; operands are evaluated from left to right. Running the
;; compiled instructions by original version will raise
;; an exception, if we don't impose an order of evaluation.
(let ([first-code (if (last-exp? seq)
(compile (first-exp seq) compile-time-env blacklist target
linkage)
(compile (first-exp seq) compile-time-env blacklist target
'next))])
(if (last-exp? seq)
first-code
(preserving '(env continue)
first-code
(compile-sequence (rest-exp seq) compile-time-env blacklist target
linkage)))))

(define (compile-lambda exp compile-time-env blacklist target linkage)
(let ((proc-entry (make-label 'entry))
(after-lambda (make-label 'after-lambda)))
(let ((lambda-linkage
(if (eq? linkage 'next) after-lambda linkage)))
(append-instruction-sequences
(tack-on-instruction-sequence
(end-with-linkage lambda-linkage
(make-instruction-sequence '(env) (list target)
`((assign ,target
(op make-compiled-
procedure)
(label ,proc-entry)
(reg env))))))

(compile-lambda-body exp
(extend-compile-time-environment
(lambda-parameters exp)
compile-time-env)
blacklist
proc-entry)
after-lambda)))))

(define (extend-compile-time-environment vars base-env)
(cons vars base-env))

(define (compile-lambda-body exp compile-time-env blacklist proc-entry)
(let ((formals (lambda-parameters exp)))
(append-instruction-sequences
(make-instruction-sequence '(env proc argl) '(env)
`,(,proc-entry
(assign env (op compiled-procedure-env) (reg proc))
(assign env
(op extend-environment)
(const ,formals)
(reg argl)
(reg env)))))

(compile-sequence
(lambda-body exp)
compile-time-env
blacklist
'val 'return)))))

(define (compile-application exp compile-time-env blacklist target linkage)
(let ((proc-code (compile (operator exp) compile-time-env blacklist 'proc 'next)))
(operand-codes
(map (lambda (operand) (compile operand compile-time-env blacklist 'val 'next))
(operands exp)))
(preserving '(env continue)
proc-code
(preserving '(proc continue)
(construct-arglist operand-codes)))

```

```

        (compile-procedure-call target linkage)))))

(define (compile-if exp compile-time-env blacklist target linkage)
  (let ((t-branch (make-label 'true-branch))
        (f-branch (make-label 'false-branch))
        (after-if (make-label 'after-if)))
    (let ((consequent-linkage
           (if (eq? linkage 'next) after-if linkage)))
      (let ((p-code (compile (if-predicate exp) compile-time-env blacklist 'val 'next))
            (c-code
              (compile
                (if-consequent exp) compile-time-env blacklist target consequent-linkage))
            (a-code
              (compile (if-alternative exp) compile-time-env blacklist target linkage)))
        (preserving '(env continue)
                    p-code
                    (append-instruction-sequences
                      (make-instruction-sequence '(val) '()
                        `((test (op false?) (reg val))
                           (branch (label ,f-branch))))
                    (parallel-instruction-sequences
                      (append-instruction-sequences t-branch c-code)
                      (append-instruction-sequences f-branch a-code)))
                  after-if)))))

;; Exercise 5.41

(define (index-of lst val)
  (define (iter n lst)
    (cond [(null? lst) -1]
          [(eq? (car lst) val) n]
          [else (iter (+ n 1) (cdr lst))]))
  (iter 0 lst))

(define (found? address)
  (not (eq? address 'not-found)))

(define (find-variable var compile-time-env)
  (define (iter f-num compile-time-env)
    (if (null? compile-time-env)
        'not-found
        (let ((d-num (index-of (first-frame compile-time-env)
                               var)))
          (if (= d-num -1)
              (iter (+ f-num 1) (enclosing-environment compile-time-env))
              (list f-num d-num)))))
  (iter 0 compile-time-env))

;; Exercise 5.42

(define (top-level-environment? compile-time-env)
  (= (length compile-time-env) 1))

(define (open-code-procedure? proc blacklist)
  (and (memq proc the-open-code-procedures)
       (not
         (memq proc blacklist)))))

(define (add-proc-to-blacklist var blacklist)
  (set-cdr! blacklist (cons var (cdr blacklist)))))

;; define a variable in the current compile-time-env
;; add a procedure to blacklist if it meets the conditions
;; (1. the current compile-time-env is top-level-environment
;; 2. it is open-code-procedure)

(define (define-compile-time-variable! var compile-time-env blacklist)
  (if (and (top-level-environment? compile-time-env)
            (open-code-procedure? var blacklist))
      (add-proc-to-blacklist var blacklist)

      (if (not (memq var (car compile-time-env)))
          (set-car! compile-time-env (cons var (car compile-time-env)))))

(define (variable-in-top-level? var compile-time-env)
  (eq?
    (frame-number (find-variable var compile-time-env))
    (- (length compile-time-env) 1)))

(define (compile-variable exp compile-time-env target linkage)
  (let ([address (find-variable exp compile-time-env)])
    (end-with-linkage linkage
                      (make-instruction-sequence '(env)
                        (list target)
                        `((assign ,target

```

```

        (op lexical-address-lookup)
        (const ,address)
        (reg env))))))
(define (compile-assignment exp compile-time-env blacklist target linkage)
  (let* ([var (assignment-variable exp)])
    [get-value-code
      (compile (assignment-value exp) compile-time-env blacklist 'val 'next)]
    [address (find-variable var compile-time-env)])
  (if (and (variable-in-top-level? var compile-time-env)
            (open-code-procedure? var blacklist))
      (add-proc-to-blacklist var blacklist))
  (end-with-linkage linkage
    (preserving '(env)
      get-value-code
      (make-instruction-sequence '(env val)
        (list target)
        `((perform (op lexical-
address-set!)
          (const ,address)
          (reg val)
          (reg env))
        (assign ,target (const
ok))))))))))

(define (compile-definition exp compile-time-env blacklist target linkage)
  (let ((var (definition-variable exp)))
    (define-compile-time-variable! var compile-time-env blacklist)
    (let ((get-value-code
          (compile (definition-value exp) compile-time-env blacklist 'val 'next)))
      (end-with-linkage linkage
        (preserving '(env)
          get-value-code
          (make-instruction-sequence '(env val) (list target)
            `((perform (op define-
variable!
          (const ,var)
          (reg val)
          (reg env))
        (assign ,target (const
ok)))))))))))

;;; setting up a compile time environment which is perfectly
;;; parallel to the runtime environment, which means we do not
;;; deal with the global environment specially (all variables
;;; in the global environment have a correspoding lexical address.
(define (setup-compile-environment)
  (list
    (frame-variables
      (first-frame
        (setup-environment)))))

;;; Exercise 5.43

;;; **NB** we redefine some function in order to support an version with scanning out.
;;; we don't define a variable in the current compile-time-env,
;;; just add a procedure to blacklist if it meets the conditions
;;; (1. the current compile-time-env is top-level-environment
;;; 2. it is open-code-procedure)
(define (define-compile-time-variable! var compile-time-env blacklist)
  (if (and (top-level-environment? compile-time-env)
            (open-code-procedure? var blacklist))
      (add-proc-to-blacklist var blacklist)))

;;; setting up a compile time environment,
;;; the top level environment of a compile time environment
;;; is empty, which means all variables
;;; in the global environment don't have a lexical address,
;;; so we need compile those expressions with lookup-variable-value.
(define (setup-compile-environment)
  '()))

;;; we modified this, since the top level environment is empty.
(define (variable-in-top-level? var compile-time-env)
  (not (found? (find-variable var compile-time-env)))))

;;; NOTE: the special handling of the global variables
(define (compile-variable exp compile-time-env target linkage)
  (let* ([address (find-variable exp compile-time-env)]
    [lookup (if (found? address) 'lexical-address-lookup 'lookup-variable-value)]
    [object (if (found? address) address exp)]
    [needed (if (found? address) '(env) ())])

```

```

(modified (if (found? address) (list target) (list target 'env)))
[maybe-change-env (if (found? address)
  '()
  `((assign env (op get-global-environment)))))

(end-with-linkage linkage
  (make-instruction-sequence needed
    modified
    (append
      maybe-change-env
      `((assign ,target
        (op ,lookup)
        (const ,object)
        (reg env))))))

(define (compile-assignment exp compile-time-env blacklist target linkage)
  (let* ([var (assignment-variable exp)]
    [get-value-code
      (compile (assignment-value exp) compile-time-env blacklist 'val 'next)]
    [address (find-variable var compile-time-env)]
    [setter (if (found? address) 'lexical-address-set! 'set-variable-value!)]
    [object (if (found? address) address var)]
    [needed (if (found? address) '(env val) '(val))]
    [modified (if (found? address) (list target) (list target 'env))]
    [maybe-change-env (if (found? address)
      '()
      `((assign env (op get-global-environment))))])
  (if (and (variable-in-top-level? var compile-time-env)
    (open-code-procedure? var blacklist))
    (add-proc-to-blacklist var blacklist))
  (end-with-linkage linkage
    (preserving '(env)
      get-value-code
      (make-instruction-sequence needed
        modified
        (append
          maybe-change-env
          `((perform (op ,setter)
            (const ,object)
            (reg val)
            (reg env))
            (assign ,target (const
              ok))))))))
  ;;; NOTE: we scan out internal definitions in lambda body here.
  (define (compile-lambda-body exp compile-time-env blacklist proc-entry)
    (let ((formals (lambda-parameters exp)))
      (append-instruction-sequences
        (make-instruction-sequence '(env proc argl) '(env)
          `,(proc-entry
            (assign env (op compiled-procedure-env) (reg proc))
            (assign env
              (op extend-environment)
              (const ,formals)
              (reg argl)
              (reg env))))
        (compile-sequence
          (scan-out-defines
            ;**
            (lambda-body exp))
          ;**
          compile-time-env
          blacklist
          'val 'return)))))

;; Exercise 5.44
(define (spread-arguments operands compile-time-env blacklist)
  (let ([arg1-code (compile (first-operand operands) compile-time-env blacklist 'arg1
    'next)])
    (rest-codes (map (lambda (op) (compile op compile-time-env blacklist 'arg2
    'next))
      (rest-operands operands))))
  (if (>= (length operands) 2)
    (cons arg1-code rest-codes)
    (error "Unsupported arity!" operands)))))

(define (compile-open-code exp compile-time-env blacklist target linkage)
  (define (compile-open-code-rest operand-codes)
    (if (null? (cdr operand-codes))
      (preserving
        `(arg1)
        (car operand-codes)
        (make-instruction-sequence
          '(arg1 arg2) ;in fact, arg2 can be omitted.
          (list target)
          `((assign ,target (op ,(operator exp)) (reg arg1) (reg arg2)))))
      (preserving

```

```

    '(arg1 env)
  (car operand-codes)
  (append-instruction-sequences
    (make-instruction-sequence
      '(arg1 arg2) ;in fact, arg2 can be omitted.
      '(arg1)
      `((assign arg1 (op ,(operator exp)) (reg arg1) (reg arg2))))
    (compile-open-code-rest (cdr operand-codes))
  )))

;; we evaluate the first operand and the second operand,
;; then assign the values to the corresponding registers sequentially.
;; By accumulating arg2 into arg1, we have the accumulation of the first
;; two operands. After that, we evaluate the third and put its value into
;; arg2. As above, we accumulate arg2 into arg1 so that we have the
;; accumulation of the first three operands. And so on, until we reach
;; the last operand of the operands, this time, we put the accumulation
;; of arg1 and arg2 into the target register.

(let ([operand-codes (spread-arguments (operands exp) compile-time-env blacklist)])
  (end-with-linkage
    linkage
    (preserving
      '(env)
      (car operand-codes)
    (compile-open-code-rest (cdr operand-codes))))))

;;; Verification Support
(load "ch5-regsim.scm")

(define (lookup-prim symbol operations)
  (let ((val (assoc symbol operations)))
    (if val
        (cadr val)
        (eval symbol)))))

;;; some expressions and its corresponding answers
(define exps
  (list

    ' (begin
        (define (factorial n)
          (if (= n 1)
              1
              (* (factorial (- n 1)) n)))
        (factorial 5))
    ;; => 120

    ' (begin (define + (lambda (x) (- 1 1)))
        +
        (+ 1))
    ;; => 0

    ' (((lambda (x y)
        (lambda (a b c d e)
          ((lambda (y z) (* x y z))
            (* a b x)
            (+ c d x))))
      3
      4)
     1 7 3 4 5
    )
    ;; => 630

    ' (let ((x 3) (y 4))
        ((lambda (a b c d e)
            (let ((y (* a b x)) (z (+ c d x)))
              (* x y z)))
          1 7 3 4 5
        )))
    ;; => 630

    ' (begin
        (define (square x) (* x x))
        (define x 4)
        (+ (square 2) x x)
      )
    ;; => 12

    ' ((lambda (+ * a b x y)
        (+ (* a x) (* b y)))
      + * 1 2 3 4)
    ;; => 11

    ' ((lambda (+ * a b x y)
        (+ (* a x) (* b y)))

```

```

        * + 1 2 3 4)
;; => 24

' (begin
  ((lambda (+ * a b x y)
    (+ (* a x) (* b y)))
   * + 1 2 3 4)
  (+ 1 3 4 5))
;; => 13

' (begin
  (set! + 1)
  +)
;; => 1

' (begin
  (define y 2)
  (define - y)
  (define + -)
  (define (f x)
    (set! x +)
    (set! + *)
    (+ x 3))
  (f 2))
;; => 6

' (begin
  (define (f x)
    (define + -)
    1)
  (f 2)
  (+ 1 1))
;; => 2

' (begin
  (define (f x)
    (set! + -)
    1)
  (f 2)
  (+ 1 1))
;; => 0

' (begin
  (define + -)
  (+ 1 1))
;; => 0

' (begin
  (define (factorial n)
    (define (iter product counter)
      (if (> counter n)
          product
          (iter (* counter product)
                (+ counter 1))))
    (iter 1 1))
  (factorial 5))
;; => 120

' (begin
  (set! + (begin
    (set! + -)
    (+ 1 1)))
  +)
;; => 0

' (begin
  (set! + (begin
    (+ 1 1)))
  +)
;; => 2

))

(define (make-blacklist)
  (list '*HEAD*))

(define (compile-and-print exp)

(define the-global-environment (setup-environment))
(define (get-global-environment)
  the-global-environment)

(define demo-machine
  (make-machine
    all-reg
    (list

```

```

(list 'get-global-environment get-global-environment))
(statements
  (compile exp
    (setup-compile-environment)
    (make-blacklist)
    'val
    'next)))))

;; Extra settings for Chez Scheme
(define new-env
  (copy-environment (scheme-environment)))

(eval '(define + '()) new-env)
(eval '(define + (eval '+ (scheme-environment))) new-env)
(eval '(define - '()) new-env)
(eval '(define - (eval '- (scheme-environment))) new-env)
(eval '(define * '()) new-env)
(eval '(define * (eval '* (scheme-environment))) new-env)
(eval '(define / '()) new-env)
(eval '(define / (eval '/ (scheme-environment))) new-env)
(eval '(define = '()) new-env)
(eval '(define = (eval '=' (scheme-environment))) new-env)

(set-register-contents! demo-machine 'env (get-global-environment))

(start demo-machine)
(pretty-print exp)
(display "; => ")
(pretty-print (get-register-contents demo-machine 'val))

(if
  (eq? (eval exp new-env)
    (get-register-contents demo-machine 'val))
  (display "pass")
  (begin
    (display "fail")
    (newline)
    (display (eval exp new-env)))))

(newline)
(newline)
)

(for-each compile-and-print exps)

;;;;;;
;;;;;OUTPUT;;;;;;
;;;;;;

(begin
  (define (factorial n)
    (if (= n 1) 1 (* (factorial (- n 1)) n)))
  (factorial 5))
;; => 120
pass

(begin (define + (lambda (x) (- 1 1))) + (+ 1))
;; => 0
pass

(((lambda (x y)
  (lambda (a b c d e)
    ((lambda (y z) (* x y z)) (* a b x) (+ c d x))))
  3
  4) 1 7 3 4 5)
;; => 630
pass

(let ([x 3] [y 4])
  ((lambda (a b c d e)
    (let ([y (* a b x)] [z (+ c d x)]) (* x y z))) 1 7 3 4 5))
;; => 630
pass

(begin
  (define (square x) (* x x))
  (define x 4)
  (+ (square 2) x x))
;; => 12
pass

((lambda (+ * a b x y) (+ (* a x) (* b y))) + * 1 2 3 4)
;; => 11
pass

((lambda (+ * a b x y) (+ (* a x) (* b y))) * + 1 2 3 4)
;; => 24

```

```

pass

(begin
  ((lambda (+ * a b x y) (+ (* a x) (* b y))) * + 1 2 3 4)
  (+ 1 3 4 5))
;; => 13
pass

(begin (set! + 1) +)
;; => 1
pass

(begin
  (define y 2)
  (define - y)
  (define + -)
  (define (f x) (set! x +) (set! + *) (+ x 3))
  (f 2))
;; => 6
pass

(begin (define (f x) (define + -) 1) (f 2) (+ 1 1))
;; => 2
pass

(begin (define (f x) (set! + -) 1) (f 2) (+ 1 1))
;; => 0
pass

(begin (define + -) (+ 1 1))
;; => 0
pass

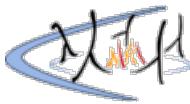
(begin
  (define (factorial n)
    (define (iter product counter)
      (if (> counter n)
          product
          (iter (* counter product) (+ counter 1))))
    (iter 1 1))
  (factorial 5))
;; => 120
pass

(begin (set! + (begin (set! + -) (+ 1 1))) +)
;; => 0
pass

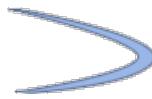
(begin (set! + (begin (+ 1 1))) +)
;; => 2
pass

;;;;;;;;;;
;;; a special test case:
;;; raises an exception in version 1
;;; works in version 2
;;; You can test it yourself!
'(begin
  (define (f)
    (define (g) h)
    (define (h) g)
    ((h)))
  (f))
;;;;;;;;;

```



# sicp-ex-5.45



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.44) | Index | Next exercise (5.46) >>

meteorgan

```
compiled:  
n      total-pushes    maximum-depth  
2        13            5  
3        19            8  
4        25            11  
so, total-pushes is 6n+1, maximum-depth is 3n-1.  
total-pushes:  
compiled/interpretation: (6n+1)/(32n-16) -> 0.1875  
special/interpretation:   (2n-2)/(32n-16) -> 0.0625  
maximum-depth:  
compiled/interpretation: (3n-1)/(5n+3) -> 0.6  
special/interpretation:   (2n-2)/(5n+3) -> 0.4
```

Rptx

A recommendation I can think of is seeing the code produced by compiling the fact procedure, we can see there are extra saves and pushes generated because compile-proc-appl declares all registers as modified even if the actual procedure does not modify any.

Maybe this procedure could be enhanced to avoid this. And declare register usage depending on the operator. With this idea I got the compiler down to 2n+1 pushes and 2n-2 depths. Just as good as the special purpose hand-coded factorial machine.

```
; You also need to modify the caller to pass the operator of the  
; expression down to compile-proc-appl  
(define safe-ops  
  '(=))  
(define (compile-proc-appl op target linkage compile-time-environment)  
  (let ((modified-regs (if (memq op safe-ops)  
                           '()  
                           all-regs)))  
    (cond ((and (eq? target 'val) (not (eq? linkage 'return)))  
           (make-instruction-sequence  
            '(proc) modified-regs  
            `((assign continue (label ,linkage))  
              (assign val (op compiled-procedure-entry)  
                  (reg proc))  
              (goto (reg val)))))  
          ((and (not (eq? target 'val))  
                (not (eq? linkage 'return)))  
           (let ((proc-return (make-label 'proc-return)))  
             (make-instruction-sequence  
              '(proc) modified-regs  
              `((assign continue (label ,proc-return))  
                (assign val (op compiled-procedure-entry)  
                    (reg proc))  
                (goto (reg val))  
                ,proc-return  
                (assign ,target (reg val))  
                (goto (label ,linkage))))))  
          ((and (eq? target 'val) (eq? linkage 'return))  
           (make-instruction-sequence  
            '(proc continue) modified-regs  
            `((assign val (op compiled-procedure-entry)  
              (reg proc))  
              (goto (reg val)))))  
          ((and (not (eq? target 'val))  
                (eq? linkage 'return))  
           (error "return linkage, target not val -- COMPILE"  
                 target))))
```

codybartfast

Part A  
=====

	Maximum Depth	Number of Pushes
Recursive Compiled	$2n + 1$	$2n + -2$
Iterative Compiled	$0n + 5$	$2n + 11$
Recursive Machine	$2n + -2$	$2n + -2$
Recursive Evaluator	$5n + 3$	$32n + -10$
Iterative Evaluator	$0n + 10$	$35n + 35$

This suggests that the compiled code would run 16 times faster, using 60% less space than purely evaluated code.

Part B  
=====

The compiler generates code that is effectively as efficient as the hand-tailored version. Presumably because the compiler implements the optimizations from this section such as open-code application of primitives.

# sicp-ex-5.46

[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]

[Edit] [Edit History]

Search:

<< Previous exercise (5.45) | Index | Next exercise (5.47) >>

Rptx

Here are the results:

Interpreted =>  $S(n) = 42 * \text{Fib}(n+1) - 30$

Special purpose => Pushes  $S(n) = 3 * \text{Fib}(n+1) - 3$  Depth  $2n - 2$

Normal compile => Pushes  $S(n) = 7 * \text{Fib}(n+1) - 2$  Depth  $2n$

Modified compiler. This is the compiler I posted in the previous exercise.

=> Pushes  $S(n) = 3 * \text{Fib}(n+1)$  Depth  $2n - 2$

As can be seen here. The modified compiler behaves the same as the special purpose machine.

codybartfast

Summary  
=====

Fibonacci	Maximum Depth	Number of Pushes
Interpreted	$5n + 3$	$56 * (\text{fib } n+1) - 34$
Compiled	$2n + 0$	$7 * (\text{fib } n+1) - 2$
Machine	$2n - 2$	$4 * (\text{fib } n+1) - 4$

Machine  
=====

The stats for (fib 4) are from the hand-sumulation in Ex 5.5

n	Fibonacci	Maximum Depth	Number of Pushes
1	1	0	0
2	1	2	4
3	2	...	...
4	3	6	16
5	5	...	...
			$4 * (\text{fib } n+1) - 4$

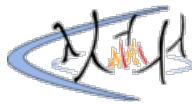
If using the optimizaton from Ex. 5.6 the number of pushes will be smaller,  
i.e.:  $3 * (\text{fib } n+1) + 3$

Interpreted & Compiled  
=====

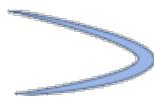
Fibonacci	Interpreted		Compiled	
	Max Depth	No of Pushes	Max Depth	No of Pushes
1	1	8	22	3
2	1	13	78	4
				12

3	2	18	134	6	19
4	3	23	246	8	33
5	5	28	414	10	54
6	8	33	694	12	89
7	13	38	1,142	14	145
8	21	43	1,870	16	236
9	34	48	3,046	18	383
10	55	53	4,950	20	621
11	89	58	8,030	22	1,006
56*(fib n+1)-34				7*(fib n+1)-2	

Last modified : 2020-03-03 16:47:23  
**WiLiKi 0.5-tekili-7** running on **Gauche 0.9**



# sicp-ex-5.47



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.46) | Index | Next exercise (5.48) >>

Rptx

```
; add a test for compound procedures.
(define (compile-procedure-call op target linkage compile-time-environment)
  (let ((primitive-branch (make-label 'primitive-branch))
        (compiled-branch (make-label 'compiled-branch))
        (compound-branch (make-label 'compound-branch)) ; ** ex 5.48
        (after-call (make-label 'after-call)))
    (let ((comp-linkage ; compiled and compound
          (if (eq? linkage 'next) after-call linkage)))
      (append-instruction-sequences
       (make-instruction-sequence
        '(proc) '()
        `((test (op primitive-procedure?) (reg proc))
          (branch (label ,primitive-branch))
          (test (op compound-procedure?) (reg proc)) ; ** 5.48
          (branch (label ,compound-branch))))
       (parallel-instruction-sequences
        (parallel-instruction-sequences ;**
         (append-instruction-sequences
          compiled-branch
          (compile-proc-appl
           op target comp-linkage compile-time-environment))
        (append-instruction-sequences ; compound branch.
         compound-branch ; compile-compound-code will call
         (compile-compound-call ; compound-application in the evaluator.
          op target comp-linkage compile-time-environment)))
       (append-instruction-sequences
        primitive-branch
        (end-with-linkage
         linkage
         (make-instruction-sequence
          '(proc argl)
          (list target)
          `((assign ,target
                    (op apply-primitive-procedure)
                    (reg proc)
                    (reg argl)))))))
      after-call)))))

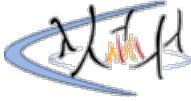
; The evaluator is arranged so that at apply-dispatch,
; the continuation would be at the top of the stack.
; So we must save the continue before passing control
; to the evaluator.

(define (compile-compound-call op target linkage compile-time-environment)
  (let ((modified-regs (if (memq op safe-ops)
                           '()
                           all-regs)))
    (cond ((and (eq? target 'val) (not (eq? linkage 'return)))
           (make-instruction-sequence
            '(proc) modified-regs
            `((assign continue (label ,linkage))
              (save continue)
              (goto (reg compapp)))))
          ((and (not (eq? target 'val))
                (not (eq? linkage 'return)))
           (let ((proc-return (make-label 'proc-return)))
             (make-instruction-sequence
              '(proc) modified-regs
              `((assign continue (label ,proc-return))
                (save continue)
                (goto (reg compapp))
                ,proc-return
                (assign ,target (reg val))
                (goto (label ,linkage)))))))
```

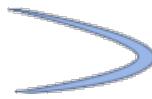
```
((and (eq? target 'val) (eq? linkage 'return))
  (make-instruction-sequence
   ' (proc continue) modified-reg
   ' ((save continue)
      (goto (reg compapp))))
  ((and (not (eq? target 'val))
        (eq? linkage 'return))
   (error "return linkage, target not val -- COMPILE"
          target))))
```

---

Last modified : 2014-09-05 17:43:30  
WiLiKi 0.5-tekili-7 running on Gauche 0.9



# sicp-ex-5.48



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

[<< Previous exercise \(5.47\) | Index | Next exercise \(5.49\) >>](#)

```

; this code will get executed by primitive-apply in ch5-eceval-compiler.scm
; specifically, by (apply-primitive-procedure)
(define (compile-and-run expression)
  (let ((stack (eceval 'stack))
        (instructions
         (assemble
          (append
           (statements (compile expression 'val 'next))

           ; print the result after executing statements
           '((goto (reg printres)))
           eceval)))

        ; get rid of old value of continue
        ; this is optional, because (initialize-stack) will
        ; clear the stack after print-result
        (stack 'pop)

        ; the next 2 commands in primitive-apply are:
        ; (restore continue)
        ; (goto (reg continue))
        ; this forces eceval to jump to and execute instructions
        ((stack 'push) instructions)))

; -----
; -----
; -----
; -----



(define (test-5.48)
  (load "ch5-compiler.scm")
  (load "load-eceval-compiler.scm"))

; add and initialize new register printres to expose label print-result
; cf. compadd from 5.47
(set! eceval (make-machine
  (cons 'printres eceval-compiler-register-list)
  eceval-operations ; procedures accessible via (op) in asm code
  (cons
    '(assign printres (label print-result))
    eceval-compiler-main-controller-text)))

; procedures accessible at the EC-Eval prompt
(append! primitive-procedures
  (list (list 'compile-and-run compile-and-run)))

  (start-eceval)
)
(test-5.48)

; ;;; EC-Eval input:
; (compile-and-run '(define (f n) (if (= n 1) 1 (* (f (- n 1)) n))))
; ;;; EC-Eval value:
; ok
; ;;; EC-Eval input:
; (f 5)
; ;;; EC-Eval value:
; 120

```

```
(define (prim-compile-and-run expression)
  (assemble (statements
              (compile expression 'val 'return '())
              eceval)))
```

```

(define (compile-and-run? exp)
  (tagged-list? exp 'compile-and-run))

(define (compile-and-run-exp exp)
  (cadadr exp))

; this is added to ev-dispatch
(test (op compile-and-run?) (reg exp))
(branch (label ev-compile-and-run))

; here is ev-compile-and-run
ev-compile-and-run
(assign val (op compile-and-run-exp) (reg exp))
(assign val (op prim-compile-and-run) (reg val))
(goto (label external-entry))

-----
;;; EC-Eval input:
(compile-and-run
 '(define (factorial n)
    (if (= n 1)
        1
        (* (factorial (- n 1)) n)))))

(total-pushes = 0 max-depth = 0)
;;; EC-Eval value:
ok

;;; EC-Eval input:
(factorial 5)

(total-pushes = 11 max-depth = 8)
;;; EC-Eval value:
120

;; from the stack data we can see that it is in fact the compiled version
;; of factorial.

```

codybartfast

To avoid having the compiler or evaluator directly access the machine/eceval, I added a (dynamic) assemble instruction to the Register Machine's assembler.

```

Compiler
=====
(define (statements-with-return exp)
  (statements
    (compile exp empty-ctenv 'val 'return)))

```

```

Register Machine - Assembler
=====

```

To keep it simple the assemble instruction doesn't take any parameters, it reads the statements from the val register, and then writes the assembled instructions back to val:

```

(define (make-assemble-val inst machine labels operations pc)
  (lambda ()
    (let* ((statements (get-register-contents machine 'val))
           (instructions
             (assemble statements machine)))
      (set-register-contents! machine 'val instructions)
      (advance-pc pc))))

```

This is called from make-execution-procedure:

```

(define (make-execution-procedure ...)
  (cond ((eq? (car inst) 'assign)
         (make-assign inst machine labels ops pc))
        ...
        ((eq? (car inst) 'assemble-val) ;*
         (make-assemble-val inst machine labels ops pc)) ;*
        (else (error \"Unknown instruction type -- ASSEMBLE\" inst))))

```

```

EC-Evaluator
=====

```

The ec-evaluator compiles the expression to val, uses assemble-val to replace the statements with the assembled instructions and then goes to those instructions:

```
compile-and-run
  (assign exp (op compile-and-run-exp) (reg exp))
  (assign val (op statements-with-return) (reg exp))
  (assemble-val)
  (goto (reg val))
```

This is called from eval-dispatch:

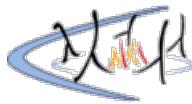
```
eval-dispatch
  (test (op self-evaluating?) (reg exp))
  (branch (label ev-self-eval))
  ...
  (test (op compile-and-run?) (reg exp)) ;*
  (branch (label compile-and-run)) ;*
  (test (op application?) (reg exp))
  (branch (label ev-application))
  (goto (label unknown-expression-type))
```

These use the following additional primitive operations:

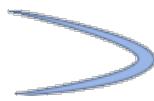
```
(list 'compile-and-run?
      (lambda (exp) (tagged-list? exp 'compile-and-run)))
(list 'statements-with-return statements-with-return)
(list 'compile-and-run-exp cadr)
```

thanhnghuyen2187

I wrote a detailed walkthrough on my blog:<https://nguyễnhuythanh.com/posts/sicp-5.48>



# sicp-ex-5.49



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.48) | Index | Next exercise (5.50) >>

- 5.48: a compiling interpreter
- 5.49: an interpreting compiler

```
ypeels

(load "ch5-regsim.scm") ; for (make-machine) and (assemble)
(load "ch5-compiler.scm") ; for (compile)
(load "ch5-eceval-support.scm") ; for syntax and utility procedures
(load "ch5-eceval-compiler.scm") ; for operation list, needed by assembler

; return linkage makes compiled code return to print-result below
(define (compile-and-assemble expr)
  (assemble
    (statements (compile expr 'val 'return))
    ec-comp-exec))

; compiler expects same register set as eceval
(define ec-comp-exec-registers
  '(expr env val proc argl continue unev))

(define ec-comp-exec-operations
  (append
    eceval-operations ; from ch5-eceval-compiler.scm
    (list (list 'compile-and-assemble compile-and-assemble)))))

(define ec-comp-exec-controller-text '(
  ; main loop same as eceval
  read-compile-exec-print-loop
  (perform (op initialize-stack))
  (perform (op prompt-for-input) (const ";; EC-Comp-Exec input:"))
  (assign expr (op read))
  (assign env (op get-global-environment))
  (assign continue (label print-result))
  (goto (label compile-and-execute)) ; ** label name changed
  print-result
  (perform (op print-stack-statistics))
  (perform (op announce-output) (const ";; EC-Comp-Exec value:"))
  (perform (op user-print) (reg val))
  (goto (label read-compile-exec-print-loop))

  ; the entirety of the new machine! as per the problem statement,
  ; all complexity is deferred to "primitives" (compile) and (assemble)
  compile-and-execute
  (assign val (op compile-and-assemble) (reg expr))
  (goto (reg val))
))

(define ec-comp-exec (make-machine
  ec-comp-exec-registers
  ec-comp-exec-operations
  ec-comp-exec-controller-text))

(define (start-ec-comp-exec)
  (set! the-global-environment (setup-environment))
  (ec-comp-exec 'start)
)

(start-ec-comp-exec)
```

codybartfast

```
Controller
=====
(define rcep-controller
  '((assign env (op get-global-environment))
    read
    (perform (op prompt-for-input) (const ";;; RCEP-RM input:"))
    (assign val (op read))
    (assign val (op compile) (reg val))
    (assemble-val)
    (assign continue (label after-execute))
    (goto (reg val))
    after-execute
    (perform (op announce-output) (const ";;; RCEP-RM value:"))
    (perform (op user-print) (reg val))
    (goto (label read))))
```

Assemble-val is the same as in the previous exercise.

Operations

=====
(define operations
 (append eceval-operations
 (list (list 'read read)
 (list 'compile statements-with-return)
 (list 'prompt-for-input prompt-for-input)
 (list 'announce-output announce-output)
 (list 'user-print user-print)))

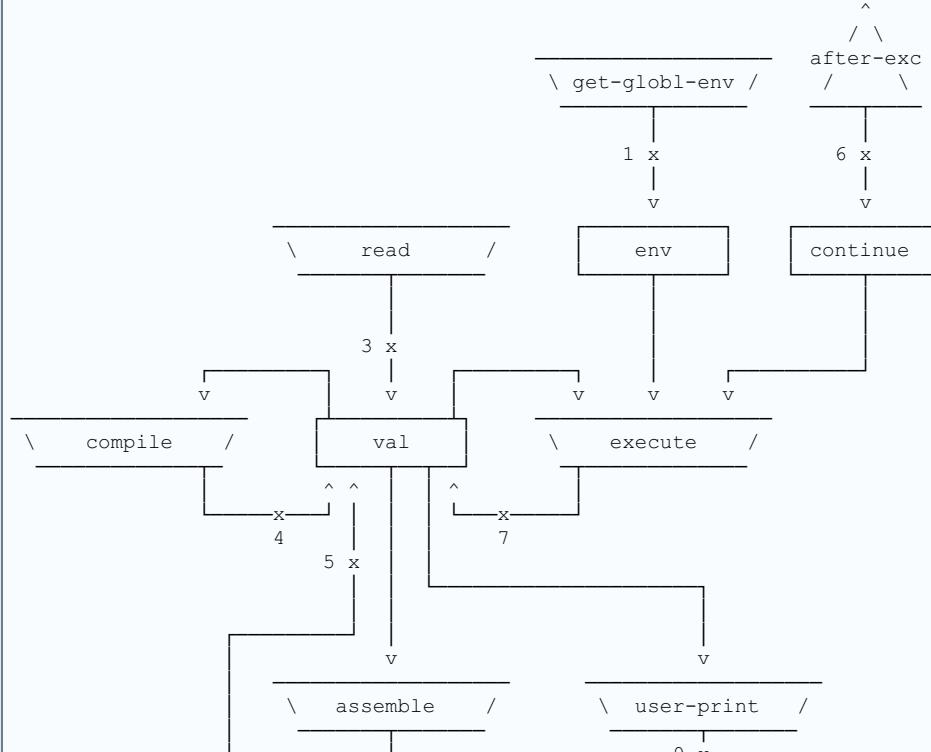
Statements-with-return is the same as in the previous exercise.

Running the Read-Compile-Execute-Print Loop

=====
(start (make-machine operations rcep-controller))

Data-Path

=====

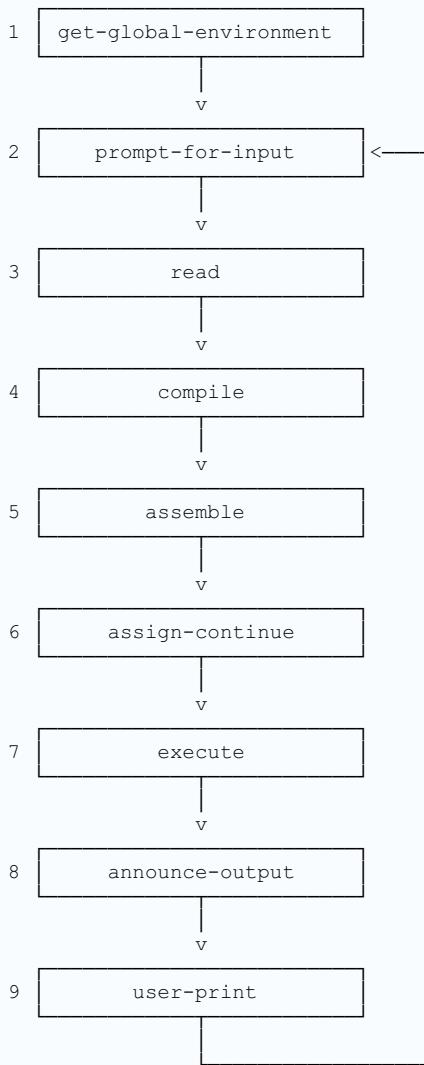


\prompt-for-inpt/  
2 x

\announce-output/  
8 x

### Controller Diagram

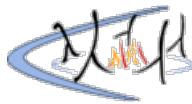
---



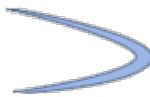
### Demo

---

```
;;; RCEP-RM input:  
(define (factorial n)  
  (if (= n 1)  
      1  
      (* (factorial (- n 1)) n)))  
  
;;; RCEP-RM value:  
ok  
  
;;; RCEP-RM input:  
(factorial 5)  
  
;;; RCEP-RM value:  
120  
  
;;; RCEP-RM input:
```



# sicp-ex-5.50



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

<< Previous exercise (5.49) | Index | Next exercise (5.51) >>

codybartfast

Metacircular Evaluator Expression  
=====

In an attempt to keep things simple I used the most basic implementation of the metacircular evaluator from Section 4.1. As the question says this needs to be put in one big begin statement, which is then quoted to provide an expression that can be compiled.

A few changes were needed to the content of the expression:

1. add a map procedure so that when map is called by the metacircular evaluator its procedure calls are evaluated at the right level (i.e., within the metacircular evaluator and not by a primitive map in the explicit-control procedures). See Exercise 4.14.
2. add any primitive-procedures that will be needed by the programs evaluated by the metacircular evaluator.
3. add (driver-loop) to the end of the metacircular evaluator so the REPL starts automatically.

Metacircular Evaluator Code:  
-----

```
(define mc-evaluator-exp
  '(begin

    (define (eval exp env)
      (cond ((self-evaluating? exp) exp)
            ((variable? exp) (lookup-variable-value exp env))
            ((quoted? exp) (text-of-quotation exp))
            ...
            ...
            ... LOADS MORE ...

            (map proc items)
            (if (null? items)
                '()
                (cons (proc (car items))
                      (map proc (cdr items)))))

    (define primitive-procedures
      (list (list '= =)
            (list '* *)
            (list '- -)
            (list 'cons cons)
            (list 'list list)
            (list 'equal? equal?)))
      ))

    ... BIT MORE ...

    (driver-loop)
  ))
```

"Explicit-Control" Procedures  
=====

We don't need the Explicit Control Evaluator itself, but we do need its primitive operations and primitive procedures.

I started with the most recent ec-evaluator (ex 5.48) and made the following changes:

1. Add primitive procedures. The explicit control's table of primitive procedures needs to include:
  - a. all primitive procedures used by the metacircular evaluator's implementation, e.g., caddr, cddr, set-car!, etc ...
  - b. any primitive procedures in the metacircular evalutors table of primitive procedures, e.g., =, \*, -, etc ...
2. Add apply-in-underlying-scheme to the explicit-control procedures and include it in the explicit-control's table of primitive-procedures.

Primitive procedures in the metacircular are double tagged. E.g., the cons procedures will be:

```
(primitive (primitive <underlying-cons>))
```

The outer tag is added by the metacircular evaluator, the inner tag is added by the ec-evaluator's primitive-procedure-objects procedure.

So the apply-in-underlying-scheme provided to the metacircular evaluator needs to remove this inner tag before passing it to the scheme that the explicit-control procedures are implemented on.

3. Removed checking around primitive procedures (Exercise 5.30). This is not functionally necessary, but it helped debugging while getting the metacircular evaluator working. Without checking the machine crashes as soon a primitive-procedure encounters a problem. With checking the machine doesn't crash until the result of the bad primitive procedure call is used. (If we want primitive checking of the interpreted code then we need to implement that in the metacircular evaluator, not in the compiled code).

"Explicit Control" Procedure Code:

---

```
(define (primitive-implementation proc) (cadr proc))

(define (apply-in-underlying-scheme proc args)
  (apply (primitive-implementation proc) args))

(define primitive-procedures
  ;; Primitives used by the metacirculator evaluator's implementation
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        ...
        (list 'set-car! set-car!)
        (list 'set-cdr! set-cdr!)
        (list 'caddr caddr)
        (list 'cdddr cdddr)
        (list 'apply-in-underlying-scheme apply-in-underlying-scheme)
        ;; Additional primitive procedures installed in the MC-evaluator
        (list '- -)
        (list '* *)
        (list 'equal? equal?)))
  ))
```

Compiler

---

Started with the most recent compiler (Ex 5.48). We do need internal definitions to work, and I think that requires having scan-out-defines installed in the compiler (Exercise 5.43). Additional changes:

1. The metaciruclar evaluator's implementation uses let. Therefore we need to add support for let to the compiler (or rewrite the metacircular evaluator replacing let statements with lambda calls). Support for let can be added to the compiler the same way it was added to the metacircular evaluator (Exercise 4.6) as a derived expression.
2. Added a convenience procedure, statements-with-next, that compiles an expression, (in this case mc-evaluator-exp), with the 'next linkage and extracts the statements.

Compiler Code:

---

```
;; install let->combination

(define (compile exp ctenv target linkage)
  (cond ((self-evaluating? exp)
         (compile-self-evaluating exp ctenv target linkage))
        ((quoted? exp) (compile-quoted exp ctenv target linkage))
        ...))
```

```

...
((cond? exp) (compile (cond->if exp) ctev target linkage))
((let? exp) (compile (let->combination exp) ctev target linkage))
...
((application? exp)
 (compile-application exp ctev target linkage))
(else
 (error "Unknown expression type -- COMPILE" exp)))))

;; let->combination

(define (let-body exp)
  (cddr exp))

(define (let-pairs exp)
  (cadr exp))

(define let-pair-id car)

(define let-pair-value cadr)

(define (let-params exp)
  (map let-pair-id
       (let-pairs exp)))

(define (let-values exp)
  (map let-pair-value
       (let-pairs exp)))

(define (let? exp) (tagged-list? exp 'let))

(define (let->combination exp)
  (make-call
    (make-lambda (let-params exp)
                 (let-body exp))
    (let-values exp)))

;; convenience procedure

(define (statements-with-next exp)
  (statements-with exp 'next))

(define (statements-with exp linkage)
  (statements
    (compile exp empty-ctev 'val linkage)))

```

Running the Metacircular Evaluator  
=====

To run the evaluator we just need to prepend an instruction to load the global environment. Because there is a call to (driver-loop) at the end of the evaluator the REPL starts automatically:

```

(define mc-eval-code
  (statements-with-next mc-evaluator-exp))

(define program
  (cons '(assign env (op get-global-environment))
        mc-eval-code))

(define machine (make-machine eceval-operations program))

(start machine)

```

Sample Output:  
-----

```

;;; M-Eval input:
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))

;;; M-Eval value:
ok

;;; M-Eval input:
(factorial 5)

;;; M-Eval value:
120

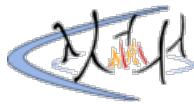
;;; M-Eval input:

```

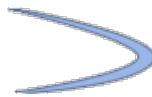
Full code is on github **SICP Chapter5**

---

Last modified : 2020-10-23 12:57:17  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-5.51



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

---

<< Previous exercise (5.50) | Index | Next exercise (5.52) >>

postboy

Attempt to solve this exercise:<https://github.com/postboy/sicp-homework/tree/master/5.51>

codybartfast

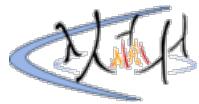
An interpreter that supports the syntax and code from the book.  
<https://github.com/codybartfast/sicp-ex5.51>

lockywolf

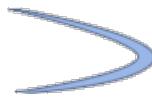
An interpreter in Fortran 2018, translated line-by-line. Added reader is naive, garbage collector is like in the book, strings are not leaking, but symbols are compared in O(n).  
<https://gitlab.com/Lockywolf/chibi-sicp/-/blob/master/index.org>

---

Last modified : 2020-11-08 10:45:27  
WiLiKi 0.5-tekili-7 running on **Gauche 0.9**



# sicp-ex-5.52



[Top Page] [Recent Changes] [All Pages] [Settings] [Categories] [Wiki Howto]  
[Edit] [Edit History]  
Search:

---

<< Previous exercise (5.51) | Index

lockywolf

Solution, compiling scheme to Fortran 2018. <https://gitlab.com/Lockywolf/chibi-sicp-/blob/master/index.org>

---

Last modified : 2020-11-08 10:49:05  
WiLiKi 0.5-tekili-7 running on Gauche 0.9