

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 + 3))` 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)
                     (min-precedence 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, `min-precedence` 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 (min-precedence 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.¹

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.

```

j=> (deriv '(x + 3 * (x + y + 2)) 'x)
;Value: 4

j=> (deriv '(x + 3) 'x)
;Value: 1

j=> (deriv '(x * y * (x + 3)) 'x)
;Value 88: ((x * y) + (y * (x + 3)))

;; Will extraneous parens throw our deriv for a loop?
j=> (deriv '((x * y) * (x + 3)) 'x)
;Value 89: ((x * y) + (y * (x + 3)))

j=> (deriv '(x * (y * (x + 3))) 'x)
;Value 90: ((x * y) + (y * (x + 3)))

```

¹ 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 `cdr`).

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.

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 derivatives 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:

$(3 + 10 * x)$? SUM 3 AND $10 * x$ | FIRST STEP

$(10 * x)$? PRODUCT 10 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.

```

(define (op? o)
  (or (eq? o '+) (eq? o '-') (eq? o '*') (eq? o '^)))

(define (precedence o)

```



```

(cond ((eq? o '+) 1)
      ((eq? o '-') 1)
      ((eq? o '* ) 2)
      ((eq? o '^ ) 3)
      (else (error "unknown operator: PRECEDENCE" o))))

(define (associativity o)
  (cond ((eq? o '+) 'left)
        ((eq? o '-') 'left)
        ((eq? o '* ) 'left)
        ((eq? o '^ ) 'right)
        (else (error "unknown operator: ASSOCIATIVITY" o))))

(define (shunting-yard exp)
  (define (apply-op output op)
    (let ((lhs (cadr output))
          (rhs (car output)))
      (cons
        (cond ((eq? op '+) (make-sum lhs rhs))
              ((eq? op '-') (make-difference lhs rhs))
              ((eq? op '* ) (make-product lhs rhs))
              ((eq? op '^ ) (make-exponentiation lhs rhs))
              (else error "unknown operator: APPLY-OP" op))
        (cddr output))))
  (define (iter output operators exp)
    (if (null? exp)
        (if (null? operators) ; pushing whatever is left in op stack into
            output
            (car output)
            (iter (apply-op output (car operators)) (cdr operators) exp))
        (let ((token (car exp)))
            (cond ((list? token) ; pushing sublist into output
                  (iter (cons (shunting-yard token) output) operators (cdr
exp)))
                  ((op? token) ; pushing new operation into output or op stack
                  (if (and (not (null? operators))
                          (or (and (eq? (associativity token) 'left)
                                   (<= (precedence token)
                                       (precedence (car operators))))
                          (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.