A LISP programming exercise

Jan L.A. van de Snepscheut
Computer Science
California Institute of Technology
Pasadena, CA 91125

14 January 1992

Keywords functional programming, program derivation.

Abstract We present the derivation of a solution to a LISP programming exercise. The derivation is in three steps. First, an inefficient solution is given. Second, the quintessence of a more efficient solution is captured in a number of equalities. Third, an efficient solution is derived from the inefficient one by a number of transformation steps, each of which is justified by the equalities.

Introduction

Given are two LISP objects. Write a boolean function that returns true just when the two arguments have the same fringe. The fringe of an object is the list of atoms in the object in their order of occurrence and ignoring the parenthesized structure in the given object. Since atom `nil` is equivalent to the empty list `()` it is to be ignored also. For example,

```
(same (a (b c)) ((a (b)) (c))) = t
```

and

```
(same (a (b c)) ((a (c)) (b))) = nil   .
```

This problem is one of the standard programming problems in LISP. It is often used to illustrate the need or attraction of new features, such as coroutines, parallel processes, or lazy evaluation. In this note we obtain an efficient solution without introducing any of these features, even though they may have their attraction. We do not emphasize the resulting program but are mainly interested in the programming process: the activity of arriving at the solution.

We give two essentially different solutions to this problem. The first solution constructs the two fringes and compares them.

```
(define
   (same (lambda (a b)
      (samefringe (fringe a) (fringe b))))
   (fringe (lambda (x)
      (cond ((null x) nil)
            ((atom x) (cons x nil))
            (t (append (fringe (car x)) (fringe (cdr x)))))))
   (samefringe (lambda (fra frb)
```

```
        (cond ((null fra) (null frb))
              ((null frb) nil)
              ((eq (car fra) (car frb))
                 (samefringe (cdr fra) (cdr frb)))
              (t nil)))))
```

Besides the standard functions **atom**, **eq**, **car**, **cdr**, and **cons**, we use **null** and **append**. The expression `(null x)` corresponds to `x=nil`, and **append** catenates its two arguments.

A more efficient solution

We can make the function `fringe` more efficient by adding an accumulating parameter and avoid `append`, but this is not what we are going to do. The second solution is based on the observation that the above solution constructs two complete fringes and only thereafter starts comparing them. For the sake of efficiency it would be much better if we could combine the two operations and, especially, stop both the comparison and construction processes if a discrepancy between the two fringes is encountered. In the worst case, in which the two fringes are equal, no benefits accrue from such a solution, but all other cases can be expected to show a reduction in execution time. Also, computing the fringes and comparing them on-the-fly reduces the storage requirements since, at any time, the parts of the two fringes that have been constructed and compared are equal and need not be stored.

The essential idea is to construct a function on an object that does not construct the entire fringe, but rather constructs its first element plus some remainder. The remainder is any structure whose fringe equals the remainder of the whole fringe. This leads directly to

```
      if (fringe a) = nil   then
[0]             (split a) = nil
      if (fringe a) ≠ nil   then
[1]             (car (fringe a)) = (car (split a))
[2]             (cdr (fringe a)) = (fringe (cdr (split a)))
```

A consequence of the above three rules is

```
[3]             (null (fringe a)) = (null (split a))
```

Instead of the function `samefringe` we need a function that compares the leading elements (if any) and, in case of equality, deals with the remainders. The function definition can be derived from the definition of `samefringe` by postulating the intended meaning:

```
        (samefringe (fringe a) (fringe b))
      = (samesplit (split a) (split b))
```

We substitute in the definition for `samefringe` and obtain

```
        (samesplit (split a) (split b))
      =     {  by postulate  }
        (samefringe (fringe a) (fringe b))
      =     {  definition of samefringe  }
        (cond ((null (fringe a)) (null (fringe b)))
              ((null (fringe b)) nil)
              ((eq (car (fringe a)) (car (fringe b)))
```

```
            (samefringe (cdr (fringe a)) (cdr (fringe b))))
         (t nil))
=      {  [3]  }
   (cond ((null (split a)) (null (split b)))
         ((null (split b)) nil)
         ((eq (car (fringe a)) (car (fringe b)))
            (samefringe (cdr (fringe a)) (cdr (fringe b))))
         (t nil))
=      {  [1]  twice  }
   (cond ((null (split a)) (null (split b)))
         ((null (split b)) nil)
         ((eq (car (split a)) (car (split b)))
            (samefringe (cdr (fringe a)) (cdr (fringe b))))
         (t nil))
=      {  [2]  }
   (cond ((null (split a)) (null (split b)))
         ((null (split b)) nil)
         ((eq (car (split a)) (car (split b)))
            (samefringe (fringe (cdr (split a)))
                        (fringe (cdr (split b))))))
         (t nil))
=      {  postulated meaning of samesplit  }
   (cond ((null (split a)) (null (split b)))
         ((null (split b)) nil)
         ((eq (car (split a)) (car (split b)))
            (samesplit (split (cdr (split a)))
                       (split (cdr (split b)))))
         (t nil))
```

Thus we have obtained the function definition for `samesplit`. This leaves us with the task of defining function `split`. We have a lot of choice here, since all we require of `split` is that it satisfies the relation to `fringe` as given above. One solution is to use `fringe` for `split`, but that defeats the whole purpose of the exercise: it is correct, but not efficient. We observe

```
   (fringe x)
=    {  definition of fringe  }
   (cond ((null x) nil)
         ((atom x) (cons x nil))
         (t (append (fringe (car x))
                    (fringe (cdr x)))))
=    {  definition of append  }
   (cond ((null x) nil)
         ((atom x) (cons x nil))
         ((null (fringe (car x))) (fringe (cdr x)))
```

```
            (t (cons (car (fringe (car x)))
                     (append (cdr (fringe (car x)))
                             (fringe (cdr x))))))
   =   { [3] }
      (cond ((null x) nil)
            ((atom x) (cons x nil))
            ((null (split (car x))) (fringe (cdr x)))
            (t (cons (car (fringe (car x)))
                     (append (cdr (fringe (car x)))
                             (fringe (cdr x))))))
   =   { [1] }
      (cond ((null x) nil)
            ((atom x) (cons x nil))
            ((null (split (car x))) (fringe (cdr x)))
            (t (cons (car (split (car x)))
                     (append (cdr (fringe (car x)))
                             (fringe (cdr x))))))
   =   { [2] }
      (cond ((null x) nil)
            ((atom x) (cons x nil))
            ((null (split (car x))) (fringe (cdr x)))
            (t (cons (car (split (car x)))
                     (append (fringe (cdr (split (car x))))
                             (fringe (cdr x))))))
   =   { (fringe (cons a b)) = (append (fringe a) (fringe b)) }
      (cond ((null x) nil)
            ((atom x) (cons x nil))
            ((null (split (car x))) (fringe (cdr x)))
            (t (cons (car (split (car x)))
                     (fringe (cons (cdr (split (car x)))
                                   (cdr x))))))
```

We are now going to define `split` by following the above formula for `fringe` and comparing it with the relation specified between the two. In the first alternative, `(fringe x)` returns `nil` and in this case `(split x)` should be `nil` also. In the second alternative, `(car (fringe x))=x` and `(cdr (fringe x))=nil`. It is required that `(car (split x))=x` and `(fringe (cdr (split x)))=nil` in this case. Since `(fringe nil)=nil` we are led to the choice `nil` for `(cdr (split x))` in this case. In the third alternative, `(fringe x)` returns `(fringe (cdr x))` so we let `(split x)` return `(split (cdr x))`. In the last alternative we have again a `cons` operation. The `car` thereof should be the same for `fringe` and `split`. The `cdr` of what `fringe` returns is of the form `(fringe` e `)` where e is a complicated expression. The requirement on `split` is that it return a value such that the `fringe` thereof is the fringe of e. Here we find ourselves in the fortunate position of being able to choose e for that value, and decide that we are done. We have thus constructed

```
    (split x)
  =
    (cond ((null x) nil)
          ((atom x) (cons x nil))
          ((null (split (car x))) (split (cdr x)))
          (t (cons (car (split (car x)))
                   (cons (cdr (split (car x))))
                         (cdr x)))))      .
```

The complete program reads as follows. It differs from the one above only in that multiple evaluation of `(split (car x))` and of `(cdr x)` is avoided.

```
(define
   (same (lambda (a b)
      (samesplit (split a) (split b))))
   (split (lambda (x)
      (cond ((null x) nil)
            ((atom x) (cons x nil))
            (t (f (split (car x)) (cdr x))))))
   (f (lambda (scarx cdrx)
      (cond ((null scarx) (split cdrx))
            (t (cons (car (scarx))
                     (cons (cdr scarx)
                           cdrx))))))
   (samesplit (lambda (spa spb)
      (cond ((null spa) (null spb))
            ((null spb) nil)
            ((eq (car spa) (car spb))
               (samesplit (split (cdr spa)) (split (cdr spb))))
            (t nil)))))
```

Conclusion

We would like to repeat that the second solution was developed for reasons of efficiency only. If the semantics of LISP had been nonstrict instead of strict, then its implementation would have been lazy. In the case of lazy evaluation our two solutions exhibit the same behavior. This is the reason why the example is often used to advocate lazy evaluation. On the other hand, LISP's eager evaluation is easier to implement efficiently. The example shows how the benefits of lazy evaluation can be obtained in a context of eager evaluation. The dual is called strictness analysis and, in general, seems to require program annotations.

The key to the second solution was to construct a function that yields the first element of a list plus a remainder. Applying the same function to this remainder yields the second element, and so on. It appears that this method can be applied to many more functions. In the case of functions with one list as argument and one list as result, no problems arise but in the case of multiple lists a choice needs to be made. It is not at all clear how this choice affects the efficiency of the resulting program and we have therefore not tried to explain the general case.

Finally, we observe that our second solution is more efficient than the first solution only if determining the first element of a fringe is essentially less work then constructing the entire fringe. An example where this assumption is invalid is

```
(cons (cons (cons nil c) b) a)    .
```

In LISP, however, these reverse lists seem are rare, and regular lists like

```
(cons a (cons b (cons nil)))
```

are prevalent.

Acknowledgement

The above derivation was prompted by questions from my CS20 students and I welcome the opportunity to express my gratitude to them. Comments by H. Peter Hofstee, Johan J. Lukkien, the Austin Tuesday Afternoon Club, and especially the referees improved the presentation.