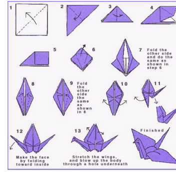
CM20167
Topic 5: Folding

Guy McCusker

1W2.1



A common pattern

We've seen lots of recursive functions on lists with the same general shape:

- for the empty list, return an answer immediately
- for a non-empty list:
  - make a recursive call on the `cdr` of the list
  - compute the result using some operation on the `car` of the list and the result of the recursive call.

Examples: length and reverse

```
(defun len (mylist) (
     if (null mylist)
       0
       (+ 1 (len (cdr mylist)))
)

(defun rev (mylist)
       (if (null mylist)
           nil
           (snoc (car mylist) (rev (cdr mylist)))
       )
)
```

Examples: length and reverse

```
(defun len (mylist) (
     if (null mylist)
        0
        (+ 1 (len (cdr mylist)))
)

(defun rev (mylist)
       (if (null mylist)
           nil
           (snoc (car mylist) (rev (cdr mylist)))
       )
)
```

Answers to return immediately when given the empty list

Examples: length and reverse

```
(defun len (mylist) (
     if (null mylist)
        0
        (+ 1 (len (cdr mylist)))
)

(defun rev (mylist)
       (if (null mylist)
           nil
           (snoc (car mylist) (rev (cdr mylist)))
       )
)
```

Recursive call on the cdr, which is then processed with the car.

Abstract away!

When there's a common pattern like this, it's worth considering whether we can *abstract away* from the concrete instances and make the pattern into a utility function.

In functional programming you can do this using higher-order functions.

The ingredients for these recursive functions are:

- the basic pattern
- the answer we give when we find an empty list
- the operation we use to combine the car with the recursive call.

A higher-order function

We want to define a function `foldr` which embodies the pattern we've observed.

In any concrete case, this function will need to know

- the answer we give when we find an empty list: call this `a`
- the operation we use to combine the `car` with the recursive call: call this `f`.

So it's a *higher-order function* which takes three arguments: the `a` and `f` we need to perform the recursion, and the list to work on.

The definition of `foldr`

```
(defun foldr (f a l)
  (if (null l)
       a
       (f (car l) (foldr f a (cdr l)))
  )
)
```

So if `l` is the list `(a1 a2 a3 ... n)`, then `(foldr f a l)` computes

`(f a1 (f a2 (f a3 (... (f an a))))).`

It's called `foldr` because the brackets nest to the right. We'll see the left-handed version later.

Length using foldr

When computing the length of a list:

- the answer for the empty list is 0
- the operation to combine the `car` and the recursive call is to ignore the `car` and just add one to the result of the recursion.

We can define a function to perform this operation:

```
(defun addone (m n) (+ 1 n))
```

Note that the argument `m`, which is going to be the `car` of the list, is not used in the body of the function.

Length using foldr

Now we can compute the length of a list `l` using a call to `foldr`:

```
user> (foldr addone 0 '(a b c d e))
5
user> (foldr addone 17 '(a b c d e))
22
```

Note what happens when we give the wrong value for the `a` argument.

Reverse using foldr

To reverse a list:

- the answer for the empty list is another empty list
- the operation to perform is `snoc`, which takes an element and a list and puts the element on the end of the list.

Assuming we've defined `snoc`, we can now reverse a list `l` with

```
(foldr snoc () l)
```

```
user> (foldr snoc () '(a b c d e))
(e d c b a)
```


Folding the other way

The folding we've seen so far is called *fold-right*. If we write the operation being performed using an infix operator like $\oplus$, then

```
(foldr ⊕ a '(a1 a2 ... an))
```

computes

$$\mathtt{a1} \oplus (\mathtt{a2} \oplus (\mathtt{a3} \oplus \cdots \oplus (\mathtt{an} \oplus \mathtt{a})))$$

and it's obvious that this is associating the operation to the right.

Folding the other way

To fold leftwards rather than rightwards means computing

$$((((\texttt{a} \oplus \texttt{a1}) \oplus \texttt{a2}) \oplus \texttt{a3}) \oplus \cdots \oplus \texttt{an})$$

Let's write the function `foldl` which does that.

Definition of `foldl`

- If the input list is empty, return `a`
- If not:
  - compute `f a (car l)`
  - keep folding this into the rest of the list.

```
(defun foldl (f a l)
  (if (null l)
      a
     (foldl f (f a (car l)) (cdr l))
     )
)
```


Analysis of `foldl`

The three arguments of `foldl` are being used in different and interesting ways:

- The third argument is the list on which we're performing recursion: the recursive call replaces `l` with `(cdr l)`
- The first argument is the operation we're folding with. This is the same throughout the recursion so it is just passed along unchanged.
- The second argument is a *partial result*: the recursive call replaces `a` with `(f a (car l))`. We pass along the value we've computed so far, and when the computation ends, return it.

Length using foldl

To compute the length of a list using `foldl` we again need an operation which adds one to one of its arguments and ignores the other.

This time we need to ignore the second argument, so the length function is:

`(foldl (lambda (m n) (+ m 1)) thelist)`:

```
user> (foldl (lambda (m n) (+ m 1)) 0 '(a b c d e))
5
user> (foldl (lambda (m n) (+ m 1)) 0 '(a b c d e f g h i j))
10
```


Reverse using foldl

Remember that `foldl` turns a list `(a1 a1 ... an)` into

$$((((\mathtt{a} \oplus \mathtt{a1}) \oplus \mathtt{a2}) \oplus \mathtt{a3}) \oplus \cdots \oplus \mathtt{an})$$

Let's think about what $\oplus$ needs to be so that this gives us

`(an ... a2 a1)`.

Reverse using foldl

The last thing this `foldl` does is to take

$$((((\mathtt{a} \oplus \mathtt{a1}) \oplus \mathtt{a2}) \oplus \mathtt{a3}) \oplus \cdots)$$

and operate on it with

$$\cdots \oplus \mathtt{an}.$$

To compute the list `(an ... a2 a1)`, it should be clear that the last operation is

`(cons an ...)`

So our operation needs to `cons` its second argument onto its first.

Reverse using `foldl`

The operation we need is therefore

```
(lambda (mylist a)
 (cons a mylist)
)
```

Let's try it out:

```
user> (foldl
       (lambda (mylist a) (cons a mylist))
       ()
       '(a b c d e))
(e d c b a)
```


Recursion versus tail-recursion

Compare the recursive calls in `foldr` and `foldl`:

```
(defun foldr (f a l)
  (if (null l)
      a
     (f (car l) (foldr f a (cdr l)))
     )
)

(defun foldl (f a l)
  (if (null l)
      a
     (foldl f (f a (car l)) (cdr l))
     )
)
```


Computing a `foldr`

Computing a `foldr` is hard work:
If the list is non-empty:

- Remember that we need to do `(f (car l) something)` and start computing the recursive call on the `cdr`.
  If the `cdr` is non-empty:
  - Remember that we need to do `(f (cadr l) something)` and start computing the recursive call on the `cddr`.
    If the `cddr` is non-empty:
    - Remember that we need to do `(f (caddr l) something)` and start computing the recursive call on the `cdddr`.
      If the `cdddr` is non-empty...
    - Recursive call complete. Compute `(f (caddr l) answer1)`
  - Recursive call complete. Compute `(f (cadr l) answer2)`
- Recursive call complete. Compute `(f (car l) answer3)`

Use of the stack

As a `foldr` is computed, the interpreter amasses a "to-do list" of pending computations.

These are handled in last-in-first-done order, that is, they form a *stack*.

This call-stack takes up a large amount of memory: much more than a simple list, for example.

It was this heavy use of the stack that caused our Sieve of Eratosthenes program to run out of space so quickly.

Computing a `foldl`

By contrast, look at how we compute a `foldl` on a list `(a1 ... an)`:

- Let's compute `(foldl f a (a1 ... an))`.
- Compute `(f a a1)`. Call it `answer1`.
- Now compute `(foldl f answer1 (a2 ... an))`.
- Compute `(f answer1 a2)`. Call it `answer2`.
- Now compute `(foldl f answer2 (a3 ... an))`.
- ...
- Now compute `(foldl f lastanswer (an))`.
- Compute `(f lastanswer an)`. That's the final answer.

There's no need for the stack here. Lisp is optimized so that when there's no "to-do list", the stack does not get used at all.

Lisp implementers are very proud of this, because they were decades ahead of other programming languages in making this optimization.

Tail recursion

The `foldl` function is called *tail recursive*: what this means is that its recursive call

```
(foldl f (f a (car l)) (cdr l))
```

invokes the recursion as the *last thing it does*. The other sub-computations

- `(f a (car l))`
- `(cdr l)`

are completed first.

Non-tail recursion

By contrast, `foldr` is not tail-recursive:

```
(f (car l) (foldr f a (cdr l)))
```

invokes the recursion `(foldr f a (cdr l))` while there's still something on the "to-do list".

Tail-recursions can always be implemented without use of the stack. Ordinary recursions cannot, unless they can first be transformed into tail recursions.

Two versions of reverse

The `foldr` version of the `reverse` function is essentially the same as the function we first thought of:

```
(defun rev (mylist)
       (if (null mylist)
           nil
           (snoc (car mylist) (rev (cdr mylist)))
       )
)
```


Two versions of reverse

On the other hand, the `foldl` version is essentially this:

```
(defun rev-accumulate (mylist done)
  (if (null mylist)
       done
      (rev-accumulate (cdr mylist) (cons (car mylist) done))
  )
)

(defun rev (mylist)
   (rev-accumulate mylist ())
)
```

`rev-accumulate` uses a second argument to *accumulate* the work done so far. Compare this with the `a` argument to `foldl`.

Making functions tail-recursive

This is a common way to turn a non-tail-recursive function into a tail-recursive one: add an extra argument, an *accumulator*, and use it to keep track of the work done so far

Here's a tail-recursive length function:

```
(defun len-accumulate (mylist seen)
  (if (null mylist)
      seen
    (len-accumulate (cdr mylist) (+ 1 seen))
    )
  )

(defun len (mylist)
  (len-accumulate mylist 0)
)
```


Factorial

Here's a function for computing the factorial of a number:

```
(defun fact (n)
  (if (zerop n)
      1
      (* n (fact (- n 1)))
      )
  )
```

It's not tail-recursive.

Exercise

Write a tail-recursive version of factorial, using an accumulator.
Can you also see how to write versions of `foldl` and `foldr` for recursion on numbers? If so, do the two versions of factorial look like uses of `foldr` and `foldl`?

Some exercises

- Write two versions of a function which adds together a list of numbers, using `foldr` and using `foldl`. Is one more efficient than the other?
- Write functions which compute the conjunction (logical AND) of a list of truth values, using `foldl` and `foldr`. How do they compare for efficiency? Remember that the binary `and` operator does not evaluate its second argument at all when the first argument is false i.e. `()`.
- What properties of addition and conjunction are you relying on to get the same answer from the two versions of these functions? (It might help to write out the folds using infix operators as we did on earlier slides.)

Some exercises

- Write tail-recursive versions of all the functions used in the Sieve of Eratosthenes and experiment to see how many more primes you can compute with the new version.
- Go and have a look at `http://www.foldr.com` and `http://www.foldl.com` and wonder why somebody pays good money to keep these sites alive.