Recursion With an Accumulator

Rob Hackman

Fall 2025

University of Alberta

Table of Contents

The problem

We want to solve `reverse` in linear time, much as we did `ascendList`

The problem

We want to solve `reverse` in linear time, much as we did `ascendList`

However, the solution we applied to `ascendList` used an observation on the arithmetic relationship between the items of the resultant `LList`. We do not have any such relationship for the elements of an arbitrary `LList` to reverse.

The problem

We want to solve `reverse` in linear time, much as we did `ascendList`

However, the solution we applied to `ascendList` used an observation on the arithmetic relationship between the items of the resultant `LList`. We do not have any such relationship for the elements of an arbitrary `LList` to reverse.

So what can be done?

Storing the answer so far

The problem, much as it was with `ascendList`, is that we want to be able to prepend items to our solution as opposed to appending them.

Storing the answer so far

The problem, much as it was with `ascendList`, is that we want to be able to prepend items to our solution as opposed to appending them.

We can solve the issue by abstracting away what we'd like to prepend to. In the past we were always operating on the recursive result, but what if we did the opposite?

Storing the answer so far

The problem, much as it was with `ascendList`, is that we want to be able to prepend items to our solution as opposed to appending them.

We can solve the issue by abstracting away what we'd like to prepend to. In the past we were always operating on the recursive result, but what if we did the opposite?

What if instead of asking the recursive call to pass the answer up to us, we instead pass the answer *down* into the recursive call!

Solving `reverse`

We will now write a recursive helper function for `reverse`, but with a new way of thinking. We will give this helper function an additional parameter, and we will make it our goal to ensure that the argument provided for this parameter will always be the *answer so far*. This means that when our base case is reached this parameter *is* our final answer!

Solving `reverse`

```
def reverseHelper(l, asf):
```

First, we identify our base case:

Solving `reverse`

```
def reverseHelper(l, asf):
```

First, we identify our base case:

Again, when `l` is empty there is nothing left to reverse. However, this time when our base case is reached what value do we produce?

Solving `reverse`

```
def reverseHelper(l, asf):
```

First, we identify our base case:

Again, when `l` is empty there is nothing left to reverse. However, this time when our base case is reached what value do we produce?

If we trust ourselves, then `asf` should store the answer so far. If there's no work left to do then the answer so far must be our final answer!

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
```

Now, we must write our recursion. Once again, stepping closer to the base case will be achieved by calculating the rest of `l`.

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), ???)
```

Now, we must write our recursion. Once again, stepping closer to the base case will be achieved by calculating the rest of `l`.

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), ???)
```

Now, we must write our recursion. Once again, stepping closer to the base case will be achieved by calculating the rest of `l`.

But what should be done to `asf` for the recursive call? We've promised ourselves that `asf` will be the answer so far. If `asf` is already the answer so far then how do we build it up for the next step?

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), ???)
```

Assume our original `LList` is of the form $(v_0, v_1, ..., v_{n-1}, v_n)$. When our function is first called no work has been done, so the answer so far should be empty ().

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), ???)
```

Assume our original `LList` is of the form $(v_0, v_1, \ldots, v_{n-1}, v_n)$. When our function is first called no work has been done, so the answer so far should be empty ().

In our first call then we must build up `asf` from the empty `LList` such that it is the result of reversing everything *before* the rest of the `LList`, because the recursion must do the work on the rest.

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), ???)
```

Assume our original `LList` is of the form $(v_0, v_1, ..., v_{n-1}, v_n)$. When our function is first called no work has been done, so the answer so far should be empty ().

In our first call then we must build up `asf` from the empty `LList` such that it is the result of reversing everything *before* the rest of the `LList`, because the recursion must do the work on the rest.

The answer then is that we must `cons` the first of our list onto the answer so far.

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), cons(first(l), asf))
```

This solution worked when we considered the *first* step of building up answer so far from the empty list in our first call to this function. What about each recursion after that?

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), cons(first(l), asf))
```

This solution worked when we considered the *first* step of building up answer so far from the empty list in our first call to this function. What about each recursion after that?

If we walk through our recursion step by step we can see that it builds up the final answer correctly!

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), cons(first(l), asf))

reverseHelper((1, 2, 3), ())
  -> reverseHelper(rest((1, 2, 3)), cons(first((1, 2, 3)), ()))
```

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), cons(first(l), asf))
```

```
reverseHelper((1, 2, 3), ())
  -> reverseHelper((2, 3), cons(first((1, 2, 3)), ()))
```

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), cons(first(l), asf))
```

```
reverseHelper((1, 2, 3), ())
  -> reverseHelper((2, 3), cons(1, ()))
```

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), cons(first(l), asf))
```

```
reverseHelper((1, 2, 3), ())
  -> reverseHelper((2, 3), (1))
```

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), cons(first(l), asf))
```

```
reverseHelper((1, 2, 3), ())
  -> reverseHelper((2, 3), (1))
   -> reverseHelper(rest((2, 3)), cons(first((2, 3)), (1)))
```

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), cons(first(l), asf))

reverseHelper((1, 2, 3), ())
  -> reverseHelper((2, 3), (1))
   -> reverseHelper((3), cons(first((2, 3)), (1)))
```

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), cons(first(l), asf))
```

```
reverseHelper((1, 2, 3), ())
  -> reverseHelper((2, 3), (1))
   -> reverseHelper((3), cons(2, (1)))
```

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), cons(first(l), asf))
```

```
reverseHelper((1, 2, 3), ())
  -> reverseHelper((2, 3), (1))
   -> reverseHelper((3), (2, 1))
```

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), cons(first(l), asf))

reverseHelper((1, 2, 3), ())
  -> reverseHelper((2, 3), (1))
   -> reverseHelper((3), (2, 1))
    -> reverseHelper(rest((3)), cons(first((3)), (2, 1)))
```

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), cons(first(l), asf))
```

```
reverseHelper((1, 2, 3), ())
  -> reverseHelper((2, 3), (1))
   -> reverseHelper((3), (2, 1))
    -> reverseHelper((), cons(first((3)), (2, 1)))
```

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), cons(first(l), asf))

reverseHelper((1, 2, 3), ())
  -> reverseHelper((2, 3), (1))
   -> reverseHelper((3), (2, 1))
    -> reverseHelper((), cons(3, (2, 1)))
```

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), cons(first(l), asf))
```

```
reverseHelper((1, 2, 3), ())
  -> reverseHelper((2, 3), (1))
   -> reverseHelper((3), (2, 1))
    -> reverseHelper((), (3, 2, 1))
```

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), cons(first(l), asf))
```

```
reverseHelper((1, 2, 3), ())
  -> reverseHelper((2, 3), (1))
   -> reverseHelper((3), (2, 1))
    -> reverseHelper((), (3, 2, 1))
     -> (3, 2, 1)
```

Solving `reverse`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), cons(first(l), asf))
```

```
reverseHelper((1, 2, 3), ())
  -> reverseHelper((2, 3), (1))
   -> reverseHelper((3), (2, 1))
    -> reverseHelper((), (3, 2, 1))
     -> (3, 2, 1)
```

And so at the final recursion when the parameter `l` is empty the parameter `asf` is the complete reversed `LList` and our solution is correct!

Completed `reverse`

Now `reverse` simply becomes a wrapper function for `reverseHelper`

```
def reverseHelper(l, asf):
  if isEmpty(l):
    return asf
  return reverseHelper(rest(l), cons(first(l), asf))

def reverse(l):
  return reverseHelper(l, empty())
```

Using an accumulator

In our answer to `reverseHelper` our parameter `asf` held the answer so far.

Using an accumulator

In our answer to `reverseHelper` our parameter `asf` held the answer so far.

As such it can be said that our final answer accumulated in the parameter `asf`.

Using an accumulator

In our answer to `reverseHelper` our parameter `asf` held the answer so far.

As such it can be said that our final answer accumulated in the parameter `asf`.

It is for this reason that this type of solution is called *recursion with an accumulator* or sometimes simply *accumulative recursion*.

Using an accumulator

In our answer to `reverseHelper` our parameter `asf` held the answer so far.

As such it can be said that our final answer accumulated in the parameter `asf`.

It is for this reason that this type of solution is called *recursion with an accumulator* or sometimes simply *accumulative recursion*.

While an accumulator may often store the answer we are building, it may also store extra information we need to complete our solution. Our previous answer to `ascendList` could also be thought of as accumulative recursion with the parameter `sub` being an accumulator!

Practice with accumulative recursion

Practice Problem: Write the function `balancedGylphs` that takes a single string parameter and returns `True` if all the parentheses `()`, brackets `[]`, and braces `{}` are *balanced* in the given string, and `False` otherwise.

We will call these collectively *enclosing glyphs*. We will call the opening kind an opening glyph, and the closing kind a closing glyph. A type of enclosing glyph is called balanced if:

- For each opening glyph there is a corresponding closing glyph that appears to the right of it.
- The number of closing glyphs is equal to the number of opening glyphs of the same type.
- The order in which closing glyphs appear matches that of the appearance of opening glyphs

Practice problem examples

Some examples for `balancedGlyphs`:

- `'{[xt(y)]z}[hello]'` is balanced, because each opening glyph has a corresponding closing glyph, and the order matches.
- `'{[(]})'` is *not* balanced, because even though each opening glyph has a corresponding closing glpyh the *order* does not match, because the closing parenthesis should come before the closing square bracket or curly brace.
- `'[Hey there :)]'` is *not* balanced, because there is a closing parenthesis without an opening parenthesis
- `'((())'` is *not* balanced, because the number of opening parentheses does not match the number of closing parentheses.