Higher-order Functions

Rob Hackman

Fall 2025

University of Alberta

Table of Contents

Doubling a `LList`

Let's write a function `doubleList` that takes a `LList` of numbers and returns a new `LList` that is the result of multiplying every number in the given `LList` by two.

```
def doubleList(l):
  if isEmpty(l):
    return empty()
  return cons(2*first(l), doubleList(rest(l)))
```

Recalling `leetSpeak`

Now, let's recall our `leetSpeak` function.

```
def leetSpeak(s):
  if s == "":
    return ""
  return convertChar(s[0]) + leetSpeak(s[1:])
```

Generalizing recursive solutions

These two functions are performing the same general task.

Generalizing recursive solutions

These two functions are performing the same general task.

- Each function is repeatedly applying some binary function f to each value of the sequence and the result of the recursion

Generalizing recursive solutions

These two functions are performing the same general task.

- Each function is repeatedly applying some binary function f to each value of the sequence and the result of the recursion
- Each function is producing a base value when the base case is reached

Folding a sequence

Both of these functions are *folding* these sequences. A *fold* of a sequence is exactly the procedure of applying a binary function f to each value of the sequence and each subsequent result of the function f.

Folding a sequence

Both of these functions are *folding* these sequences. A *fold* of a sequence is exactly the procedure of applying a binary function f to each value of the sequence and each subsequent result of the function f.

Particularly, given a sequence $S = (v_0, v_1, ..., v_{n-1}, v_n)$ and some binary function f these functions are both performing the procedure:

Folding a sequence

Both of these functions are *folding* these sequences. A *fold* of a sequence is exactly the procedure of applying a binary function f to each value of the sequence and each subsequent result of the function f.

Particularly, given a sequence $S = (v_0, v_1, ..., v_{n-1}, v_n)$ and some binary function f these functions are both performing the procedure:

$f(v_0,$

Folding a sequence

Both of these functions are *folding* these sequences. A *fold* of a sequence is exactly the procedure of applying a binary function f to each value of the sequence and each subsequent result of the function f.

Particularly, given a sequence $S = (v_0, v_1, ..., v_{n-1}, v_n)$ and some binary function f these functions are both performing the procedure:

$$f(v_0,$$
$$\qquad f(v_1,$$

Folding a sequence

Both of these functions are *folding* these sequences. A *fold* of a sequence is exactly the procedure of applying a binary function f to each value of the sequence and each subsequent result of the function f.

Particularly, given a sequence $S = (v_0, v_1, ..., v_{n-1}, v_n)$ and some binary function f these functions are both performing the procedure:

$$f(v_0,$$
$$\quad f(v_1,$$
$$\quad\quad f(v_2,$$

Folding a sequence

Both of these functions are *folding* these sequences. A *fold* of a sequence is exactly the procedure of applying a binary function f to each value of the sequence and each subsequent result of the function f.

Particularly, given a sequence $S = (v_0, v_1, ..., v_{n-1}, v_n)$ and some binary function f these functions are both performing the procedure:

$$f(v_0, \quad f(v_1, \quad f(v_2, \quad ...$$

Folding a sequence

Both of these functions are *folding* these sequences. A *fold* of a sequence is exactly the procedure of applying a binary function f to each value of the sequence and each subsequent result of the function f.

Particularly, given a sequence $S = (v_0, v_1, ..., v_{n-1}, v_n)$ and some binary function f these functions are both performing the procedure:

$$f(v_0,$$
$$f(v_1,$$
$$f(v_2,$$
$$...$$
$$f(v_{n-1},$$

Folding a sequence

Both of these functions are *folding* these sequences. A *fold* of a sequence is exactly the procedure of applying a binary function f to each value of the sequence and each subsequent result of the function f.

Particularly, given a sequence $S = (v_0, v_1, ..., v_{n-1}, v_n)$ and some binary function f these functions are both performing the procedure:

$$f(v_0, \quad f(v_1, \quad f(v_2, \quad ... \quad f(v_{n-1}, \quad f(v_n, ???)))))...$$

Folding a sequence

$$f(v_0,\ f(v_1,\ f(v_2,\ \ldots\ f(v_{n-1},\ f(v_n, ???))))))\ldots$$

What value is used for the second argument when the fold is applying the given function to the final value of the sequence?

Folding a sequence

$$f(v_0,\ f(v_1,\ f(v_2,\ \ldots\ f(v_{n-1},\ f(v_n, ???))))))\ldots$$

What value is used for the second argument when the fold is applying the given function to the final value of the sequence?

In the case of our recursions this would be the value we produce when the base case is reached!

Folding a sequence

$$f(v_0, f(v_1, f(v_2, \ldots f(v_{n-1}, f(v_n, base))))\ldots)$$

What value is used for the second argument when the fold is applying the given function to the final value of the sequence?

In the case of our recursions this would be the value we produce when the base case is reached!

Finding f

One may argue that both our functions `leetSpeak` and `doubleList` do not apply just a single function f.

Finding f

One may argue that both our functions `leetSpeak` and `doubleList` do not apply just a single function f.

For example, one may argue `leetSpeak` both calls the function `convertChar` and uses the append operation to build the final result from the recursive result.

Finding f

One may argue that both our functions `leetSpeak` and `doubleList` do not apply just a single function f.

For example, one may argue `leetSpeak` both calls the function `convertChar` and uses the append operation to build the final result from the recursive result.

However while that may be the case as we've written it we can easily rewrite both of these functions so that their recursive result is simply the result of applying some function to their first value and their recursive result.

Rewriting `leetSpeak`

Consider the following rewrite of `leetSpeak`

```
def lc(c, s):
  return convertChar(c) + s

def leetSpeak(s):
  if s == "":
    return ""
  return lc(s[0], leetSpeak(s[:1]))
```

Rewriting `leetSpeak`

Consider the following rewrite of `leetSpeak`

```
def lc(c, s):
  return convertChar(c) + s

def leetSpeak(s):
  if s == "":
    return ""
  return lc(s[0], leetSpeak(s[:1]))
```

Now our function `leetSpeak` is simply the result of folding `lc` over a string!

Rewriting `doubleList`

Consider the following rewrite of `doubleList`

```
def dc(num, l):
  return cons(2*num, l)

def doubleList(l):
  if isEmpty(l):
    return empty()
  return dc(first(l), doubleList(rest(l)))
```

Rewriting `doubleList`

Consider the following rewrite of `doubleList`

```
def dc(num, l):
  return cons(2*num, l)

def doubleList(l):
  if isEmpty(l):
    return empty()
  return dc(first(l), doubleList(rest(l)))
```

Now our function `doubleList` is simply the result of folding `dc` over a `LList` of numbers!

Visualizing `leetSpeak`

Consider the function call `leetSpeak("great")` and how it would be evaluated.

```
lc("g",
```

Visualizing `leetSpeak`

Consider the function call `leetSpeak("great")` and how it would be evaluated.

```
lc("g",
          lc("r",
```

Visualizing `leetSpeak`

Consider the function call `leetSpeak("great")` and how it would be evaluated.

```
lc("g",
        lc("r",
                lc("e",
```

Visualizing `leetSpeak`

Consider the function call `leetSpeak("great")` and how it would be evaluated.

```
lc("g",
        lc("r",
                lc("e",
                        lc("a",
```

Visualizing `leetSpeak`

Consider the function call `leetSpeak("great")` and how it would be evaluated.

```
lc("g",
        lc("r",
                lc("e",
                        lc("a",
                                lc("t", ""))))))
```

Visualizing `leetSpeak`

Consider the function call `leetSpeak("great")` and how it would be evaluated.

```
lc("g",
        lc("r",
                lc("e",
                        lc("a",
                                "7"))))
```

Visualizing `leetSpeak`

Consider the function call `leetSpeak("great")` and how it would be evaluated.

```
lc("g",
        lc("r",
                lc("e",
                        lc("a", "7"))))
```

Visualizing `leetSpeak`

Consider the function call `leetSpeak("great")` and how it would be evaluated.

```
lc("g",
        lc("r",
                lc("e",
                        "47")))
```

Visualizing `leetSpeak`

Consider the function call `leetSpeak("great")` and how it would be evaluated.

```
lc("g",
          lc("r",
                    lc("e", "47")))
```

Visualizing `leetSpeak`

Consider the function call `leetSpeak("great")` and how it would be evaluated.

```
lc("g",
        lc("r",
                "347")))
```

Visualizing `leetSpeak`

Consider the function call `leetSpeak("great")` and how it would be evaluated.

```
lc("g",
          lc("r", "347"))
```

Visualizing `leetSpeak`

Consider the function call `leetSpeak("great")` and how it would be evaluated.

```
lc("g",
          "r347")
```

Visualizing `leetSpeak`

Consider the function call `leetSpeak("great")` and how it would be evaluated.

```
lc("g", "r347")
```

Visualizing `leetSpeak`

Consider the function call `leetSpeak("great")` and how it would be evaluated.

```
"gr347"
```

Visualizing `doubleList`

Now consider the function call `doubeList(LC(1, 2, 3, 4, 5))` and how it would be evaluated.

```
dc(1,
```

Visualizing `doubleList`

Now consider the function call `doubeList(LC(1, 2, 3, 4, 5))` and how it would be evaluated.

```
dc(1,
       dc(2,
```

Visualizing `doubleList`

Now consider the function call `doubeList(LC(1, 2, 3, 4, 5))` and how it would be evaluated.

```
dc(1,
        dc(2,
                dc(3,
```

Visualizing `doubleList`

Now consider the function call `doubeList(LC(1, 2, 3, 4, 5))` and how it would be evaluated.

```
dc(1,
        dc(2,
                dc(3,
                        dc(4,
```

Visualizing `doubleList`

Now consider the function call `doubeList(LC(1, 2, 3, 4, 5))` and how it would be evaluated.

```
dc(1,
        dc(2,
                dc(3,
                        dc(4,
                                dc(5, ())))))
```

Visualizing `doubleList`

Now consider the function call `doubeList(LC(1, 2, 3, 4, 5))` and how it would be evaluated.

```
dc(1,
       dc(2,
              dc(3,
                     dc(4,
                            (10))))))
```

Visualizing `doubleList`

Now consider the function call `doubeList(LC(1, 2, 3, 4, 5))` and how it would be evaluated.

```
dc(1,
       dc(2,
              dc(3,
                     dc(4, (10)))))
```

Visualizing `doubleList`

Now consider the function call `doubeList(LC(1, 2, 3, 4, 5))` and how it would be evaluated.

```
dc(1,
       dc(2,
              dc(3,
                     (8, 10))))
```

Visualizing `doubleList`

Now consider the function call `doubeList(LC(1, 2, 3, 4, 5))` and how it would be evaluated.

```
dc(1,
        dc(2,
                dc(3, (8, 10))))
```

Visualizing `doubleList`

Now consider the function call `doubeList(LC(1, 2, 3, 4, 5))` and how it would be evaluated.

```
dc(1,
        dc(2,
                (6, 8, 10))))
```

Visualizing `doubleList`

Now consider the function call `doubeList(LC(1, 2, 3, 4, 5))` and how it would be evaluated.

```
dc(1,
        dc(2, (6, 8, 10)))
```

Visualizing `doubleList`

Now consider the function call `doubeList(LC(1, 2, 3, 4, 5))` and how it would be evaluated.

```
dc(1,
       (4, 6, 8, 10))
```

Visualizing `doubleList`

Now consider the function call `doubeList(LC(1, 2, 3, 4, 5))` and how it would be evaluated.

```
dc(1, (4, 6, 8, 10))
```

Visualizing `doubleList`

Now consider the function call `doubeList(LC(1, 2, 3, 4, 5))` and how it would be evaluated.

```
(2, 4, 6, 8, 10)
```

Visualizing `doubleList`

Now consider the function call `doubeList(LC(1, 2, 3, 4, 5))` and how it would be evaluated.

```
(2, 4, 6, 8, 10)
```

By rewriting both functions and examining how they evaluate it is clear they are both performing exactly the same procudure (the fold) and the only ways they differ are their binary function and their base value!

Visualizing `doubleList`

Now consider the function call `doubeList(LC(1, 2, 3, 4, 5))` and how it would be evaluated.

```
(2, 4, 6, 8, 10)
```

By rewriting both functions and examining how they evaluate it is clear they are both performing exactly the same procudure (the fold) and the only ways they differ are their binary function and their base value!

Now that we've observed these similarities how can we take advantage of them?

Table of Contents

First class values

In programming, and particularly functional programming, we define a concept known as *first class functions* to discuss the behaviours of some programming languages.

First class values

In programming, and particularly functional programming, we define a concept known as *first class functions* to discuss the behaviours of some programming languages.

We will define a programming language to have functions as first class values if the following conditions are true:

First class values

In programming, and particularly functional programming, we define a concept known as *first class functions* to discuss the behaviours of some programming languages.

We will define a programming language to have functions as first class values if the following conditions are true:

- Functions may be used as arguments in function calls[1]

[1] A corollary of this is that functions may be bound to identifiers

First class values

In programming, and particularly functional programming, we define a concept known as *first class functions* to discuss the behaviours of some programming languages.

We will define a programming language to have functions as first class values if the following conditions are true:

- Functions may be used as arguments in function calls[1]
- Functions may be the return value of a function call

Python does have first class functions!

[1] A corollary of this is that functions may be bound to identifiers

Table of Contents

Higher-order functions

We now know that functions that take functions themselves as the values to substitute for their parameters, or even produce a function as their return value!

Functions that operate on functions (as their parameters or their return value) are called *higher-order functions*.

We will now define our first higher-order function! This function will be the one that abstracts away the differences between functions like `leetSpeak` and `doubleList`.

Defining the `fold` operation

We have now defined the behaviour of folding over a sequence. For now we will focus just on folding `LLists` and write a function `fold` that takes an `LList`, a function `combine`, and a base value `base` and performs the fold operation we defined earlier.

Defining the `fold` operation

We have now defined the behaviour of folding over a sequence. For now we will focus just on folding `LLists` and write a function `fold` that takes an `LList`, a function `combine`, and a base value `base` and performs the fold operation we defined earlier.

```
def fold(l, combine, base):
```

It is important to note that the *type* of `combine` here is a *function*!

Defining the `fold` operation

We have now defined the behaviour of folding over a sequence. For now we will focus just on folding `LLists` and write a function `fold` that takes an `LList`, a function `combine`, and a base value `base` and performs the fold operation we defined earlier.

```
def fold(l, combine, base):
```

It is important to note that the *type* of `combine` here is a *function*!

Now, we define our base case. For an `LList` this will be when it is empty, and when that is the case we should produce the `base` value we were given

Defining the `fold` operation

We have now defined the behaviour of folding over a sequence. For now we will focus just on folding `LLists` and write a function `fold` that takes an `LList`, a function `combine`, and a base value `base` and performs the fold operation we defined earlier.

```
def fold(l, combine, base):
  if isEmpty(l):
    return base
```

Now, we need only to apply the `combine` operation to both the first of our `LList` and the recursive result.

Defining the `fold` operation

We have now defined the behaviour of folding over a sequence. For now we will focus just on folding `LLists` and write a function `fold` that takes an `LList`, a function `combine`, and a base value `base` and performs the fold operation we defined earlier.

```
def fold(l, combine, base):
  if isEmpty(l):
    return base
  return combine(first(l), ???)
```

But what is the recursive result in the case of fold?

Defining the `fold` operation

We have now defined the behaviour of folding over a sequence. For now we will focus just on folding `LLists` and write a function `fold` that takes an `LList`, a function `combine`, and a base value `base` and performs the fold operation we defined earlier.

```
def fold(l, combine, base):
  if isEmpty(l):
    return base
  return combine(first(l), ???)
```

But what is the recursive result in the case of fold?

The result of folding the operation over the rest of the `LList`!

Defining the `fold` operation

We have now defined the behaviour of folding over a sequence. For now we will focus just on folding `LLists` and write a function `fold` that takes an `LList`, a function `combine`, and a base value `base` and performs the fold operation we defined earlier.

```
def fold(l, combine, base):
  if isEmpty(l):
    return base
  return combine(first(l), fold(rest(l), combine, base))
```

Function specifications for higher-order function

Let us write the function specification for `fold`

Function specifications for higher-order function

Let us write the function specification for `fold`

```
def fold(l, combine, base):
    '''
    fold folds the function combine over the LList l
         with the base value acting as our final
         second operand.

    l       - LList
    combine -
    base    -
    returns -
    '''
```

How do we define the type of `combine`?

A notation for function types

We will choose to represent the *type* of a function as the types of its parameters and return types.

For example, we would write the type of a function that takes one string and one integer parameter and returns a float as the following `(str int -> float)`.

The same as the way the text `LList` means the type `LList`, the text enclosed in the parentheses above means the type of "a function that takes a string and integer parameter and returns a float".

Function specifications for higher-order function

Now that we know how to write the type of a function, can we complete the specification for `fold`?

Function specifications for higher-order function

Now that we know how to write the type of a function, can we complete the specification for `fold`?

```
def fold(l, combine, base):
    '''
    fold folds the function combine over the LList l with the
         base value acting as our final second operand.

    l       - LList
    combine -
    base    -
    returns -
    '''
```

What is the type of the function `combine` though? We know it takes two parameters, but what types are they? What is its return type?

Function specifications for higher-order function

Now that we know how to write the type of a function, can we complete the specification for `fold`?

```
def fold(l, combine, base):
    '''
    fold folds the function combine over the LList l with the
        base value acting as our final second operand.

    l       - LList
    combine - (any any -> any)
    base    -
    returns -
    '''
```

We could try writing `any`, to indicate that `combine` can operate on anything.

Function specifications for higher-order function

Now that we know how to write the type of a function, can we complete the specification for `fold`?

```
def fold(l, combine, base):
    '''
    fold folds the function combine over the LList l with the
         base value acting as our final second operand.

    l       - LList
    combine - (any any -> any)
    base    - any
    returns -
    '''
```

Then what is our base type? It is one of the values operated on by `combine`, so it must also be `any`?

Function specifications for higher-order function

Now that we know how to write the type of a function, can we complete the specification for `fold`?

```
def fold(l, combine, base):
    '''
    fold folds the function combine over the LList l with the
         base value acting as our final second operand.

    l       - LList
    combine - (any any -> any)
    base    - any
    returns - any
    '''
```

What about our return type? It ultimately is the value produced by `combine`, so it must also be `any`?

Using our `fold` function

Let's now try using our `fold` function.

Using our `fold` function

Let's now try using our `fold` function.

First let us try `fold` with the `dc` function we wrote!

Using our `fold` function

Let's now try using our `fold` function.

First let us try `fold` with the `dc` function we wrote!

```
fold(LL(1, 2, 3, 4, 5), dc, empty()) -> (2, 4, 6, 8, 10)
```

Using our `fold` function

Let's now try using our `fold` function.

Consider the following `fold` application — what does it do?

Using our `fold` function

Let's now try using our `fold` function.

Consider the following `fold` application — what does it do?

```
def add(x, y):
  return x + y

fold(LL(1, 2, 3, 4, 5), add, 0)
```

Using our `fold` function

Let's now try using our `fold` function.

Consider the following `fold` application — what does it do?

```
def add(x, y):
  return x + y

fold(LL(1, 2, 3, 4, 5), add, 0)
```

This computes the summation of a `LList`, all in one line!

Using our `fold` function

Let's now try using our `fold` function.

Consider the following `fold` application — what does it do?

Using our `fold` function

Let's now try using our `fold` function.

Consider the following `fold` application — what does it do?

```
def mul(x, y):
  return x*y

fold(LL(1, 2, 3, 4, 5), mul, 1)
```

Using our `fold` function

Let's now try using our `fold` function.

Consider the following `fold` application — what does it do?

```
def mul(x, y):
  return x*y

fold(LL(1, 2, 3, 4, 5), mul, 1)
```

This computes the produce of a `LList`, all in one line!

Bad usage of `fold`

Now, we try the following use of our function `fold`

```
def sc(x, y):
  # x       - a character
  # y       - an int
  # returns - a character
  return chr(ord(x)-y)

fold(LL("h", "e", "y"), sc, 3)
```

Bad usage of `fold`

Now, we try the following use of our function `fold`

```
def sc(x, y):
  # x       - a character
  # y       - an int
  # returns - a character
  return chr(ord(x)-y)

fold(LL("h", "e", "y"), sc, 3)
```

When we try this we get an error!

```
TypeError: unsupported operand type(s)
  for -: 'int' and 'str'
```

Bad usage of `fold`

Now, we try the following use of our function `fold`

```
def sc(x, y):
  # x       - a character
  # y       - an int
  # returns - a character
  return chr(ord(x)-y)

fold(LL("h", "e", "y"), sc, 3)
```

When we try this we get an error!

```
TypeError: unsupported operand type(s)
  for -: 'int' and 'str'
```

What is the issue?

Bad usage of `fold`

Now, we try the following use of our function `fold`

```
def sc(x, y):
  # x       - a character
  # y       - an int
  # returns - a character
  return chr(ord(x)-y)

fold(LL("h", "e", "y"), sc, 3)
```

The issue is that the result of our `fold` becomes:

```
sc("h", sc("e", sc("y", 3)))
```

Bad usage of `fold`

Now, we try the following use of our function `fold`

```
def sc(x, y):
  # x       - a character
  # y       - an int
  # returns - a character
  return chr(ord(x)-y)

fold(LL("h", "e", "y"), sc, 3)
```

The issue is that the result of our `fold` becomes:

```
sc("h", sc("e", "v"))
```

But `sc` cannot use a string as its second parameter! However, we didn't violate the specification we wrote for our `fold` because we said these could all be `any`! So clearly our specification is wrong!

Fixing the `fold` specification

When defining higher-order functions it is often the case that there will be a relationship between the type of the function we're operating on and our other values. We can use free variables in our types to denote types that can be `any`, but have some relationship to other types in our specification!

```
def fold(l, combine, base):
    '''
    fold folds the function combine over the LList l with the
         base value acting as our final second operand.

    l       - LList
    combine - (X Y -> Z)
    base    - any
    returns - anu
    '''
```

Fixing the `fold` specification

```
def fold(l, combine, base):
    '''
    fold folds the function combine over the LList l with the
        base value acting as our final second operand.

    l       - LList
    combine - (X Y -> Z)
    base    - any
    returns - anu
    '''
```

We will begin by changing all the values in `combine` to free variables, and then working out the relationships.

Fixing the `fold` specification

```
def fold(l, combine, base):
    '''
    fold folds the function combine over the LList l with the
         base value acting as our final second operand.

    l       - LList
    combine - (X Y -> Z)
    base    - any
    returns - anu
    '''
```

We will begin by changing all the values in `combine` to free variables, and then working out the relationships.

The `X`, `Y`, and, `Z` here are free variables that represent "some" type.

Fixing the `fold` specification

```
def fold(l, combine, base):
    '''
    fold folds the function combine over the LList l with the
        base value acting as our final second operand.

    l       - LList
    combine - (X Y -> Z)
    base    - Y
    returns - any
    '''
```

We know that the base value is used as the second argument to the call to combine. As such, the type of base should match with Y.

Fixing the `fold` specification

```
def fold(l, combine, base):
    '''
    fold folds the function combine over the LList l with the
        base value acting as our final second operand.

    l       - LList
    combine - (X Y -> Z)
    base    - Y
    returns - Z
    '''
```

Furthermore, we know that `fold` simply returns the result of `combine` in the recursive case. That means the return type of `fold` must be `Z`.

Fixing the `fold` specification

```
def fold(l, combine, base):
    '''
    fold folds the function combine over the LList l with the
         base value acting as our final second operand.

    l       - LList
    combine - (X Y -> Z)
    base    - Y
    returns - Z
    '''
```

Does this catch the problem from the above fold of our function `sc` though?

Fixing the `fold` specification

```
def fold(l, combine, base):
    '''
    fold folds the function combine over the LList l with the
         base value acting as our final second operand.

    l       - LList
    combine - (X Y -> Z)
    base    - Y
    returns - Z
    '''
```

No it doesn't! The problem from `sc` was that `sc` returned a string, and the result of `sc` was then used as the second argument to *itself*! However `sc` expected an integer as its second argument so it didn't work!

Fixing the `fold` specification

```
def fold(l, combine, base):
    '''
    fold folds the function combine over the LList l with the
         base value acting as our final second operand.

    l       - LList
    combine - (X Y -> Y)
    base    - Y
    returns - Y
    '''
```

Since the return value of `combine` is going to be used as its own second argument these types must also match!

Fixing the `fold` specification

```
def fold(l, combine, base):
    '''
    fold folds the function combine over the LList l with the
        base value acting as our final second operand.

    l       - LList
    combine - (X Y -> Y)
    base    - Y
    returns - Y
    '''
```

Since the return value of `combine` is going to be used as its own second argument these types must also match!

This means each of `combines` second parameter and return type, `base`, and the return type of `fold` are all the same type!

Fixing the `fold` specification

```
def fold(l, combine, base):
    '''
    fold folds the function combine over the LList l with the
         base value acting as our final second operand.

    l       - LList of X
    combine - (X Y -> Y)
    base    - Y
    returns - Y
    '''
```

Lastly, since each element of `l` is used as the first argument to `combine` that means that `l` must actually be a `LList of X`

`foldr` and the `cmput274` module

This function we've defined is actually known as `foldr` and performs what is known as a *right fold*.

`foldr` and the `cmput274` module

This function we've defined is actually known as `foldr` and performs what is known as a *right fold*.

A more sophisticated `foldr` is available for you already in the `cmput274` module. Unlike the one we wrote together on the slides here it works on both `LLists` and strings, as well as other built-in Python sequences.

`foldr` and the `cmput274` module

This function we've defined is actually known as `foldr` and performs what is known as a *right fold*.

A more sophisticated `foldr` is available for you already in the `cmput274` module. Unlike the one we wrote together on the slides here it works on both `LLists` and strings, as well as other built-in Python sequences.

For example:

```
foldr("great", lc, "") -> 'gr347'
```

Understanding `foldr`

Knowledge Check: Consider the following function definition

```
def sub(x, y):
  return x - y
```

What is the result of the function call `foldr(LL(7, 10, 1, 22), sub, 0)`? Figure out the result first by hand and then check your result by executing the code. If your answer does not match try to figure out where you went wrong!

Practicing with `foldr`

Practice Problem: Write the function `parity` which takes a `BinaryStr` a parameter and returns `True` if the number of ones in the string is even, and `False` if the number of ones in the string is odd. Your function should not use explicit recursion and instead should use `foldr` to achieve any repetition necessary.

We define a `BinaryStr` as:

- The empty string ""
- "0" + BinaryStr
- "1" + BinaryStr

As always you may write any helper functions that you like (at least one helper function will be necessary — the argument for `foldr`!)

More practice with `foldr`

Practice Problem: We have previously solved the problem of reversing a `LList`. Write a function `foldReverse` which solves takes a single `LList` parameter and returns the reverse of that `LList`. However, once again you may not use explicit recursion and must use only `foldr` if you require repetition.