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:

$f(v_0,$

$\quad f(v_1,$

$\qquad f(v_2,$

$\qquad\quad ...$

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:

$f(v_0,$

$\quad f(v_1,$

$\quad\quad f(v_2,$

$\quad\quad\quad ...$

$\quad\quad\quad\quad f(v_{n-1},$

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:

$$
\begin{array}{l}
f(v_0, \\
\qquad f(v_1, \\
\qquad\qquad f(v_2, \\
\qquad\qquad\qquad ... \\
\qquad\qquad\qquad\quad f(v_{n-1}, \\
\qquad\qquad\qquad\qquad\quad f(v_n, ???))))))...
\end{array}
$$

Folding a sequence

$$
\begin{array}{l}
f(v_0, \\
\qquad f(v_1, \\
\qquad\qquad f(v_2, \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad\quad f(v_{n-1}, \\
\qquad\qquad\qquad\qquad\quad f(v_n, ???))))))\ldots
\end{array}
$$

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

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

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

$$
\begin{array}{l}
f(v_0, \\
\quad\quad f(v_1, \\
\quad\quad\quad\quad f(v_2, \\
\quad\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad\quad\quad f(v_{n-1}, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad f(v_n, base))))))\ldots
\end{array}
$$

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

```python
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

```python
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`

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`

For both of the following questions you should not use explicit recursion, and should instead only use `foldr` for necessary repetition. Note that the bodies of both functions should really just be to return the result of a call to `foldr` — the real work comes in discovering and writing the correct function to fold!

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`.

Practice Problem: Write a function `maxLList` that takes a single parameter that is a `LList` of numbers and returns the largest number in the `LList`.

Considering accumulative recursion

While `foldr` is very powerful in abstracting away the simple recursions we wrote early on, it doesn't aim to solve problems like `reverse` which we solved with accumulative recursion.

Considering accumulative recursion

While `foldr` is very powerful in abstracting away the simple recursions we wrote early on, it doesn't aim to solve problems like `reverse` which we solved with accumulative recursion.

Can we abstract away the differences between accumulative recursive functions in order to build a higher-order function that replicates their behaviour?

Considering accumulative recursion

While `foldr` is very powerful in abstracting away the simple recursions we wrote early on, it doesn't aim to solve problems like `reverse` which we solved with accumulative recursion.

Can we abstract away the differences between accumulative recursive functions in order to build a higher-order function that replicates their behaviour?

We can try, first we need to identify the commonalities and differences between accumulative recursive solutions.

Finding the last word

We are going to write a function `lastPalindrome` that takes a `LList` of strings and returns the string with the largest index that is a palindrome. That is, it returns the *last* string in order in the `LList` that is a palindrome. If no palindromes exist in the `LList` then it returns `False`.

For example, if the `LList` `("hello", "racecar", "trap", "kayak", "foo")` was used as an argument the function would produce `"kayak"` and not `"racecar"` as `"kayak"` is the *last* string in the `LList` that is a palindrome.

While we can write `lastPalindrome` without accumulative recursion, accumulative recursion is a more natural way to find the last occurrence of something, as we simply have to update our accumulator with each instance we find as we recurse deeper through our `LList`. So, we will write a helper function `lastPalindromeAcc` which is the accumulative recursion solution to the problem.

```
def lastPalindromeAcc(l, lastSoFar):
```

```
def lastPalindromeAcc(l, lastSoFar):
```

Now, we must decide on our base case. Again, the empty `LList` is a good option. Once again as with most accumulative recursions we can simply produce our accumulator when we've reached the base case.

```
def lastPalindromeAcc(l, lastSoFar):
  if isEmpty(l):
    return lastSoFar
```

Now, what must we do in our recursive case?

```
def lastPalindromeAcc(l, lastSoFar):
  if isEmpty(l):
    return lastSoFar
```

Now, what must we do in our recursive case?

It depends on whether our current string is a palindrome, so we should first determine that.

```
def lastPalindromeAcc(l, lastSoFar):
  if isEmpty(l):
    return lastSoFar
```

Now, what must we do in our recursive case?

It depends on whether our current string is a palindrome, so we should first determine that.

We can use string slicing to easily calculate the reverse of a string, and equality to check if it is a palindrome.

```
def lastPalindromeAcc(l, lastSoFar):
  if isEmpty(l):
    return lastSoFar
  if first(l) == first(l)[::-1]:
    ...
  else:
    ...
```

Regardless of whether the current string is a palindrome or not we must recurse on the rest of the `LList` in case there is a later palindrome.

```
def lastPalindromeAcc(l, lastSoFar):
  if isEmpty(l):
    return lastSoFar
  if first(l) == first(l)[::-1]:
    return lastPalindromeAcc(rest(l), ???)
  else:
    return lastPalindromeAcc(rest(l), ???)
```

Regardless of whether the current string is a palindrome or not we must recurse on the rest of the LList in case there is a later palindrome.

The question then is simply what to do with our accumulator in each case.

```
def lastPalindromeAcc(l, lastSoFar):
  if isEmpty(l):
    return lastSoFar
  if first(l) == first(l)[::-1]:
    return lastPalindromeAcc(rest(l), ???)
  else:
    return lastPalindromeAcc(rest(l), ???)
```

By the nature of the order in which we recurse, if we have found a palindrome then it appears *later* than any we have already seen. As such, we can replace `lastSoFar` with this new palindrome. If this string is not a palindrome then we simply ignore it and our last seen palindrome remains the same.

```
def lastPalindromeAcc(l, lastSoFar):
  if isEmpty(l):
    return lastSoFar
  if first(l) == first(l)[::-1]:
    return lastPalindromeAcc(rest(l), first(l))
  else:
    return lastPalindromeAcc(rest(l), lastSoFar)
```

By the nature of the order in which we recurse, if we have found a palindrome then it appears *later* than any we have already seen. As such, we can replace `lastSoFar` with this new palindrome. If this string is not a palindrome then we simply ignore it and our last seen palindrome remains the same.

```
def lastPalindrome(l):
  return lastPalindromeAcc(l, False)
```

Finally, we must write the actual `lastPalindrome` function that wraps `lastPalindromeAcc`. What value should we use for initializing `lastSoFar` though?

Finally, we must write the actual `lastPalindrome` function that wraps `lastPalindromeAcc`. What value should we use for initializing `lastSoFar` though?

If no palindrome is ever found then the initial value is the value that will be kept as `lastSoFar` the entire recursion. As such it is clear we must initialize `lastSoFar` to the value that should be produced when no palindrome is found — `False`.

Comparing `lastPalindrome` and `reverse`

At first glance the viewer of `lastPalindromeAcc` and `reverseHelper` may think they are not all that similar!

```
def lastPalindromeAcc(l, lastSoFar):
  if isEmpty(l):
    return lastSoFar
  if first(l) == first(l)[::-1]:
    return lastPalindromeAcc(rest(l), first(l))
  else:
    return lastPalindromeAcc(rest(l), lastSoFar)

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

Comparing `lastPalindrome` and `reverse`

At first glance the viewer of `lastPalindromeAcc` and `reverseHelper` may think they are not all that similar!

```
def lastPalindromeAcc(l, lastSoFar):
  if isEmpty(l):
    return lastSoFar
  if first(l) == first(l)[::-1]:
    return lastPalindromeAcc(rest(l), first(l))
  else:
    return lastPalindromeAcc(rest(l), lastSoFar)

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

But, they are! We just need to abstract out the differences!

A supposition on accumulative recursion

We constructed `foldr` when we observed we could construct arbitrarily complex functions to represent the binary operation to fold over our sequences.

A supposition on accumulative recursion

We constructed `foldr` when we observed we could construct arbitrarily complex functions to represent the binary operation to fold over our sequences.

We then make the supposition that the same can be true for simple accumulative recursions with one accumulator. We simply need to find the right binary function to fold!

Rewriting `lastPalindromeAcc`

How can we rewrite the function `lastPalindromeAcc` so that it looks more similar to `reverseHelper`?

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

Rewriting `lastPalindromeAcc`

How can we rewrite the function `lastPalindromeAcc` so that it looks more similar to `reverseHelper`?

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

That is, we want a solution that when the base case is reached simply returns the accumulator, and when the base case is *not* reached simply returns the recursive call, replacing the accumulator with some function applied to the current item and the accumulator.

Rewriting lastPalindromeAcc

How can we rewrite the function `lastPalindromeAcc` so that it looks more similar to `reverseHelper`?

```
def lastPalindromeAcc(l, acc):
  if isEmpty(l):
    return acc
  return lastPalindromeAcc(rest(l), lpCombine(first(l), asf))
```

More simply put, can we find a definition for `lpCombine` so that the above code performs the same operation as our original `lastPalindromeAcc`?

Rewriting lastPalindromeAcc

How can we rewrite the function lastPalindromeAcc so that it looks more similar to reverseHelper?

```
def lastPalindromeAcc(l, acc):
  if isEmpty(l):
    return acc
  return lastPalindromeAcc(rest(l), lpCombine(first(l), asf))
```

More simply put, can we find a definition for lpCombine so that the above code performs the same operation as our original lastPalindromeAcc?

Yes we can!

Finding `lpCombine`

```
def lpCombine(???, ???):
```

To find our definition for `lpCombine` we start by defining what our parameters will be.

Finding lpCombine

```
def lpCombine(???, ???):
```

To find our definition for `lpCombine` we start by defining what our parameters will be.

We observe that the way `lpCombine` is called is with the first of our `LList` being our first argument, and the accumulator as our second. So we should define the function as such.

```
def lpCombine(s, acc):
```

The next thing we must determine is what exactly our function should return! We note that the return value of the function is being used in `lastPalindromeAcc` as the new value of our accumulator for the next recursive call. As such, this function needs to determine what the next accumulator should be and return it.

Finding `lpCombine`

```python
def lpCombine(s, acc):
```

The next thing we must determine is what exactly our function should return! We note that the return value of the function is being used in `lastPalindromeAcc` as the new value of our accumulator for the next recursive call. As such, this function needs to determine what the next accumulator should be and return it.

But we already wrote code to determine the next accumulator in `lastPalindromeAcc`, so it is much the same here!

```
def lpCombine(s, acc):
  if s == s[::-1]:
    return s
  return acc
```

If s is a palindrome then it is our most recent palindrome! On the other hand if it is not, then we should continue to use acc.

Finding `lpCombine`

```
def lpCombine(s, acc):
  if s == s[::-1]:
    return s
  return acc
```

If `s` is a palindrome then it is our most recent palindrome! On the other hand if it is not, then we should continue to use `acc`.

This definition of `lpCombine` in combination with the new definition of `lastPalindromeAcc` yields the same behaviour we were targetting.

Comparing the new `lastPalindrome` and `reverse`

Comparing these two functions now we see they look almost identical!

```
def lastPalindromeAcc(l, acc):
  if isEmpty(l):
    return acc
  return lastPalindromeAcc(rest(l), lpCombine(first(l), asf))

def lastPalindrome(l):
  return lastPalindromeAcc(l, False)
```

Comparing the new `lastPalindrome` and `reverse`

Comparing these two functions now we see they look almost identical!

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

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

Differences between `lastPalindrome` and `reverse`

If we consider the differences between `lastPalindrome` and `reverse` we can see that there are only two ways they differ:

Differences between `lastPalindrome` and `reverse`

If we consider the differences between `lastPalindrome` and `reverse` we can see that there are only two ways they differ:

- The function that is folded over the sequence

Differences between `lastPalindrome` and `reverse`

If we consider the differences between `lastPalindrome` and `reverse` we can see that there are only two ways they differ:

- The function that is folded over the sequence
- The value to initialize acc to.

But that's the same as what we found when developing `foldr` — so how do these functions differ from those we abstracted with `foldr`?

Differences between `lastPalindrome` and `reverse`

If we consider the differences between `lastPalindrome` and `reverse` we can see that there are only two ways they differ:

- The function that is folded over the sequence
- The value to initialize acc to.

But that's the same as what we found when developing `foldr` — so how do these functions differ from those we abstracted with `foldr`?

To answer that we must trace out these functions much as we did with `doubleList` and `leetSpeak`

Consider now applying the function `lastPalindrome` to the `LList` `("hello", "racecar", "trap", "kayak", "foo")` and how it would be evaluated. For space, we shorten the name of `lpCombine` to `lpc`.

```
lpc("hello", False)
```

Visualizing `lastPalindrome`

Consider now applying the function `lastPalindrome` to the `LList` `("hello", "racecar", "trap", "kayak", "foo")` and how it would be evaluated. For space, we shorten the name of `lpCombine` to `lpc`.

```
lpc("racecar",
                lpc("hello", False))
```

Visualizing `lastPalindrome`

Consider now applying the function `lastPalindrome` to the `LList` `("hello", "racecar", "trap", "kayak", "foo")` and how it would be evaluated. For space, we shorten the name of `lpCombine` to `lpc`.

```
lpc("trap",
          lpc("racecar",
                     lpc("hello", False)))
```

Visualizing `lastPalindrome`

Consider now applying the function `lastPalindrome` to the `LList` `("hello", "racecar", "trap", "kayak", "foo")` and how it would be evaluated. For space, we shorten the name of `lpCombine` to `lpc`.

```
lpc("foo",
        lpc("trap",
                lpc("racecar",
                        lpc("hello", False))))
```

Visualizing `lastPalindrome`

Consider now applying the function `lastPalindrome` to the `LList` `("hello", "racecar", "trap", "kayak", "foo")` and how it would be evaluated. For space, we shorten the name of `lpCombine` to `lpc`.

```
lpc("foo",
          lpc("trap",
                     lpc("racecar",
                                   lpc("hello", False))))
```

However an observation can be made here as well — we don't need this large chain to be built up! We note that at each step both arguments of `lpc` are able to evaluated to a final value! More on that, and it's importance, later.

Visualizing `lastPalindrome`

Consider now applying the function `lastPalindrome` to the `LList` `("hello", "racecar", "trap", "kayak", "foo")` and how it would be evaluated. For space, we shorten the name of `lpCombine` to `lpc`.

```
lpc("foo",
          lpc("trap",
                     lpc("racecar",
                                  lpc("hello", False))))
```

However an observation can be made here as well — we don't need this large chain to be built up! We note that at each step both arguments of `lpc` are able to evaluated to a final value! More on that, and it's importance, later.

As such, a more accurate visualization would evaluate each `lpc` application immediately, replacing each call with the next call on the result of the previous one.

Visualizing `reverse`

Consider now applying the function `reverse` to the `LList` `(1, 2, 3, 4)` and how it would be evaluated.

```
cons(1, empty())
```

Visualizing `reverse`

Consider now applying the function `reverse` to the `LList` `(1, 2, 3, 4)` and how it would be evaluated.

```
cons(2,
          cons(1, empty()))
```

Visualizing `reverse`

Consider now applying the function `reverse` to the `LList` `(1, 2, 3, 4)` and how it would be evaluated.

```
cons(3,
        cons(2,
                cons(1, empty())))
```

Visualizing `reverse`

Consider now applying the function `reverse` to the `LList` `(1, 2, 3, 4)` and how it would be evaluated.

```
cons(4,
         cons(3,
                  cons(2,
                           cons(1, empty()))))
```

Generalizing the procedure

We can see from `reverse` and `lastPalindrome` a pattern emerging that defines the procedure we are trying to abstract. Given a sequence of values of the form $(v_0, v_1, ..., v_{n-1}, v_n)$ and binary function f and a base value b the procedure we are attempting to define is

$$f(v_0, base)$$

Generalizing the procedure

We can see from `reverse` and `lastPalindrome` a pattern emerging that defines the procedure we are trying to abstract. Given a sequence of values of the form $(v_0, v_1, ..., v_{n-1}, v_n)$ and binary function f and a base value b the procedure we are attempting to define is

$$f(v_1,$$
$$\quad f(v_0, base))$$

Generalizing the procedure

We can see from `reverse` and `lastPalindrome` a pattern emerging that defines the procedure we are trying to abstract. Given a sequence of values of the form $(v_0, v_1, ..., v_{n-1}, v_n)$ and binary function f and a base value b the procedure we are attempting to define is

$$
\begin{array}{l}
\quad ... \\
\qquad\quad f(v_1, \\
\qquad\qquad\qquad f(v_0, base))...
\end{array}
$$

Generalizing the procedure

We can see from `reverse` and `lastPalindrome` a pattern emerging that defines the procedure we are trying to abstract. Given a sequence of values of the form $(v_0, v_1, ..., v_{n-1}, v_n)$ and binary function f and a base value b the procedure we are attempting to define is

$$
\begin{aligned}
&f(v_{n-1},\\
&\qquad\qquad ...\\
&\qquad\qquad\qquad f(v_1,\\
&\qquad\qquad\qquad\qquad\quad f(v_0, base))...)
\end{aligned}
$$

Generalizing the procedure

We can see from `reverse` and `lastPalindrome` a pattern emerging that defines the procedure we are trying to abstract. Given a sequence of values of the form $(v_0, v_1, ..., v_{n-1}, v_n)$ and binary function f and a base value b the procedure we are attempting to define is

$$
\begin{aligned}
&f(v_n,\\
&\qquad f(v_{n-1},\\
&\qquad\qquad ...\\
&\qquad\qquad\quad f(v_1,\\
&\qquad\qquad\qquad f(v_0, base))...))
\end{aligned}
$$

Defining left folds

The procedure we've been trying to define is known as a *left* fold, as opposed to the *right* fold we already defined.

Defining left folds

The procedure we've been trying to define is known as a *left* fold, as opposed to the *right* fold we already defined.

The names left and right fold come from the side of the sequence they begin applying operations to.

Defining left folds

The procedure we've been trying to define is known as a *left* fold, as opposed to the *right* fold we already defined.

The names left and right fold come from the side of the sequence they begin applying operations to.

- The first function application that left fold evaluates is between the first element, i.e. the left-most one, and the base value.

Defining left folds

The procedure we've been trying to define is known as a *left* fold, as opposed to the *right* fold we already defined.

The names left and right fold come from the side of the sequence they begin applying operations to.

- The first function application that left fold evaluates is between the first element, i.e. the left-most one, and the base value.
- The first function application that right fold evaluates is between the last element, i.e. the right-most one, and the base value.

Defining left folds

The procedure we've been trying to define is known as a *left* fold, as opposed to the *right* fold we already defined.

The names left and right fold come from the side of the sequence they begin applying operations to.

- The first function application that left fold evaluates is between the first element, i.e. the left-most one, and the base value.
- The first function application that right fold evaluates is between the last element, i.e. the right-most one, and the base value.

As such the name of the function we want to define will be `foldl`.

Defining `foldl`

```
def foldl(l, f, acc):
    '''
    foldl performs a left fold of the function f
          over l with the base value  supplied in acc
          acting as our first second operand.

    l       - LList of X
    f       - (X Y -> Y)
    acc     - Y
    returns - Y
    '''
```

Our parameter types for `foldl` are the same as they were for `foldr`. The major difference is that our third parameter ceases just to be our base value, but instead will be our accumulator that must be initialized to our base value when `foldl` is called.

Defining `foldl`

```
def foldl(l, f, acc):
  '''
  foldl performs a left fold of the function f
        over l with the base value  supplied in acc
        acting as our first second operand.

  l       - LList of X
  f       - (X Y -> Y)
  acc     - Y
  returns - Y
  '''
```

First, our base case is the same as it was for both our accumulative recursive examples — simply return the accumulator if the base case has been reached.

Defining `foldl`

```
def foldl(l, f, base):
  if isEmpty(l):
    return acc
```

First, our base case is the same as it was for both our accumulative recursive examples — simply return the accumulator if the base case has been reached.

Defining `foldl`

```
def foldl(l, f, base):
  if isEmpty(l):
    return acc
```

Our recursive case now simply becomes recursively calling `foldl` on the `rest` of our `LList` and updating our accumulator to be the result of applying `f` to the current first value of our list and the accumulator.

Defining `foldl`

```
def foldl(l, f, base):
  if isEmpty(l):
    return acc
  return foldl(rest(l), f, f(first(l), base))
```

Our recursive case now simply becomes recursively calling `foldl` on the `rest` of our `LList` and updating our accumulator to be the result of applying `f` to the current first value of our list and the accumulator.

Reversing a `LList` with `foldl`

Now that we have `foldl` we can reverse a `LList` simply by performing a left fold of the function `cons` over a `LList`! Consider:

```
foldl(LL(1, 2, 3, 4), cons, LL()) -> (4, 3, 2, 1)
```

`foldl` and the `cmput274` module

The function `foldl` is already defined for us in the `cmput274` module, so you have no need to copy this definition into your own programs if you want to use `foldl`.

Additionally, the module provided version provides support for arbitrary `Python` sequences and not just `LLists`.

Understanding `foldl`

Knowledge Check: Consider the following function definition

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

What is the result of the function call `foldl(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!

Does the result differ from the same expression but using `foldr` instead of `foldl`? If so why?

Practice Question: Write a Python function `extremes` that has a single non-empty `LList` of numbers as its parameter `l`. The function returns a `LList` with exactly two elements in it, the first of which is the *minimum* element of `l`, and the second element is the *maximum* element of `l`.

You may not use any explicit recursion in solving the question, and must use `foldl` for any necessary repetition.

Hint: As with writing any `fold` application the difficulty of this question lies in finding the function to fold over the `LList` and the value to initialize your `base` value to.

Practice writing higher-order functions

Practice Question: A common process we need to perform in computing is to *map* a function onto a sequence. Given a unary function f and a sequence S of the form $(v_0, v_1, ..., v_{n-1}, v_n)$ the result of mapping f onto S is the sequence:

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

Write the function `map` that takes two parameters, the first being a unary function and the second being a `LList`. Your function should return the result of mapping the given function onto the given `LList`.

Examples are given on the following slide.

Examples of `map`

```
def add3(x):
  return x + 3

def double(x):
  return x*2
```

Consider the two following definitions about when considering the example `map` applications below

```
map(add3, LL(0, 10, -2)) -> (3, 13, -6)
map(double, LL(0, 10, -2)) -> (0, 20, -4)
```

More practice with higher-order functions

Practice Question: Another common process we need to perform is that of *filtering* a sequence. Given a unary function f that returns a `bool` and a sequence S of the form $(v_0, v_1, ..., v_{n-1}, v_n)$ then the result of filtering S by the predicate f is the sequence:

$$S^{'} = (v_i, v_{i+1}, ...)$$

Where an element $v_i \in S^{'}$ if and only if $f(v_i) \rightarrow True$.

Write the function `filter` as defined above. Examples are given on the next slide.

Examples of `filter`

```
def isNegative(x):
  return x < 0

def isShort(s):
  return len(s) < 6
```

Consider the two following definitions about when considering the example `map` applications below

```
filter(isNegative, LL(-10, 0, 10, -2)) -> (-10, -2)
filter(isShort, LL("abc", "zaboomafoo", "great"))
    -> ("abc", "great")
```

Table of Contents

Abstracting function creation

Imagine you are writing software for image editing. To do so you define a data type `Pixel`. Your `Pixel` data type is defined as follows:

```
A Pixel is a:
- LL(CV, CV, CV)
```

And you also define a `ColorValue`, or `CV` for short, as and integer in the range $[0, 255]$.

So a `Pixel` is a `LList` of three integers in the range $[0, 255]$.

RGB values

The three values in your `Pixel` data type represent the *red*, *blue*, and *green* component of a colour. That is, the `LList` of a `Pixel` is of the form (R, G, B).

For example, the Safety Orange colour used in our "Fun on Functions" slides had the RGB value $(255, 103, 0)$.

A picture can be represented by many `Pixels` in a `LList`.

Colour shifting

Imagine as you are writing your image editing software you need to add a feature to “red shift” an image by adding 30 to the red components of each `Pixel` in your image.

Colour shifting

Imagine as you are writing your image editing software you need to add a feature to "red shift" an image by adding 30 to the red components of each `Pixel` in your image.

You observe that this is as simple as mapping a function onto your `LList` of `Pixels` so you write the function to map.

Colour shifting

Imagine as you are writing your image editing software you need to add a feature to "red shift" an image by adding 30 to the red components of each `Pixel` in your image.

You observe that this is as simple as mapping a function onto your `LList` of `Pixels` so you write the function to map.

```
def redShift(p):
  newRed = first(p) + 30
  if newRed > 255:
    return cons(255, rest(p))
  return cons(newRed, rest(p))
```

We require more shifting functions

Let’s say after completing the “red shift” feature your boss comes back and tells you to add “even redder shift” functionality to the program, which adds 60 to the red components of each `Pixel` in your image.

We require more shifting functions

Let's say after completing the "red shift" feature your boss comes back and tells you to add "even redder shift" functionality to the program, which adds 60 to the red components of each `Pixel` in your image.

You could go back and write a near identical function that just swap the value 60 for the value 30 in your code.

We require more shifting functions

Let's say after completing the "red shift" feature your boss comes back and tells you to add "even redder shift" functionality to the program, which adds 60 to the red components of each `Pixel` in your image.

You could go back and write a near identical function that just swap the value 60 for the value 30 in your code.

But what about when your boss comes back and asks for an "even more redder shift" function. What about one for green? and blue?

Abstracting function creation

It seems silly to have to write several very small and almost identical functions.

Abstracting function creation

It seems silly to have to write several very small and almost identical functions.

For example why write one function for red shifting by a value of 30, another for 60, and another for 90?

Abstracting function creation

It seems silly to have to write several very small and almost identical functions.

For example why write one function for red shifting by a value of 30, another for 60, and another for 90?

Instead wouldn't it be better to write a function that is paramaterized by not only the `Pixel` to shift but also the amount by which to shift it? This way we would only have to write *one* function for performing a red shift and simply provide different arguments for the amount to shift by!

Parameterized `redShift`

Consider the updated `redShift` function, that is paramaterized now by the shift value.

Parameterized `redShift`

Consider the updated `redShift` function, that is paramaterized now by the shift value.

```
def redShift(p, sAmt):
  newRed = first(p) + sAmt
  if newRed > 255:
    return cons(255, rest(p))
  return cons(newRed, rest(p))
```

Parameterized redShift

Consider the updated `redShift` function, that is paramaterized now by the shift value.

```
def redShift(p, sAmt):
  newRed = first(p) + sAmt
  if newRed > 255:
    return cons(255, rest(p))
  return cons(newRed, rest(p))
```

This function is a nice idea... but

Parameterized `redShift`

Consider the updated `redShift` function, that is paramaterized now by the shift value.

```
def redShift(p, sAmt):
  newRed = first(p) + sAmt
  if newRed > 255:
    return cons(255, rest(p))
  return cons(newRed, rest(p))
```

This function is a nice idea... but We can't use this function with `map`! `map` expects a unary function — this is a binary function!

Parameterized `redShift`

Consider the updated `redShift` function, that is paramaterized now by the shift value.

```
def redShift(p, sAmt):
  newRed = first(p) + sAmt
  if newRed > 255:
    return cons(255, rest(p))
  return cons(newRed, rest(p))
```

This function is a nice idea... but We can't use this function with `map`! `map` expects a unary function — this is a binary function!

If only we had a way to abstract way the differences between our various shift functions, so that we don't have to create several almost identical unary functions!

The solution to our problem

Recall that higher-order functions are those functions that operate *on* functions — this is not only functions whose parameters are functions, but also functions whose *return values* are functions!

The solution to our problem

Recall that higher-order functions are those functions that operate *on* functions — this is not only functions whose parameters are functions, but also functions whose *return values* are functions!

With our current knowledge of defining functions we could only write functions that produce a function we've already written — not a very helpful application. In order for functions that return functions to be useful, we need some way for those functions to create new functions when they are called.

The solution to our problem

Recall that higher-order functions are those functions that operate *on* functions — this is not only functions whose parameters are functions, but also functions whose *return values* are functions!

With our current knowledge of defining functions we could only write functions that produce a function we've already written — not a very helpful application. In order for functions that return functions to be useful, we need some way for those functions to create new functions when they are called.

But how can we create a new function "on-the-fly" for our higher-order functions to return?

How to create functions

There are two main ways in which we'll be able to create new functions within existing functions.

How to create functions

There are two main ways in which we'll be able to create new functions within existing functions.

- Through *lambda* functions

How to create functions

There are two main ways in which we'll be able to create new functions within existing functions.

- Through *lambda* functions
- Through *inner*[2] functions

[2]Often called *local* functions or `nested` functions in other languages.

How to create functions

There are two main ways in which we'll be able to create new functions within existing functions.

- Through *lambda* functions
- Through *inner*[2] functions

We will start by focusing on `lambda` functions.

[2]Often called *local* functions or `nested` functions in other languages.

What are lambda functions

The name for lambda functions comes from the *lambda calculus* from which the definition originates[3].

[3] Though functions in this calculus are much simpler than those we use here.

What are lambda functions

The name for lambda functions comes from the *lambda calculus* from which the definition originates[3].

A lambda function is an *anonymous* function, which simply means that it does not have to bound to an identifier. When we learned about expressions we learned about literals. The same way that 5 is an integer literal, a lambda function is a function literal.

[3] Though functions in this calculus are much simpler than those we use here.

What are lambda functions

The name for lambda functions comes from the *lambda calculus* from which the definition originates[3].

A lambda function is an *anonymous* function, which simply means that it does not have to bound to an identifier. When we learned about expressions we learned about literals. The same way that 5 is an integer literal, a lambda function is a function literal.

Most programming languages that support lambda functions allow for lambda functions to be any arbitrary function. In Python the body of a lambda function must be a single expression.

[3]Though functions in this calculus are much simpler than those we use here.

Anatomy of a Python lambda

A lambda function in Python is an expression of the form

```python
lambda x, y : x + y
```

Anatomy of a Python lambda

A lambda function in Python is an expression of the form

Keyword to begin lambda expression

```
lambda x, y : x + y
```

Anatomy of a Python lambda

A lambda function in Python is an expression of the form

Keyword to begin lambda expression

Parameter List

```
lambda x, y : x + y
```

Anatomy of a Python lambda

A lambda function in Python is an expression of the form

Keyword to begin lambda expression

Parameter List

```
lambda x, y : x + y
```

Ends Parameter List

Anatomy of a Python lambda

A lambda function in Python is an expression of the form

Keyword to begin lambda expression

Parameter List

```
lambda x, y : x + y
```

Ends Parameter List

Function body expression

Using lambda expressions

Since a lambda expression is an expression that evaluates to a function, they can be used anywhere functions can be used.

Using lambda expressions

Since a lambda expression is an expression that evaluates to a function, they can be used anywhere functions can be used.

That means lambda expressions can be used as arguments to our higher-order functions! For example, consider the following expression

Using lambda expressions

Since a lambda expression is an expression that evaluates to a function, they can be used anywhere functions can be used.

That means lambda expressions can be used as arguments to our higher-order functions! For example, consider the following expression

```
foldr(LL(5, -10, 8), lambda x, y: x + 2*y, 0) -> 17
```

Using lambda expressions

Since a lambda expression is an expression that evaluates to a function, they can be used anywhere functions can be used.

That means lambda expressions can be used as arguments to our higher-order functions! For example, consider the following expression

```
foldr(LL(5, -10, 8), lambda x, y: x + 2*y, 0) -> 17
```

Here we have folded the function that adds its first parameter to two times its second parameter over the LList $(1, 2, 3)$ with a base value of 0. The evaluation of this is:

Using lambda expressions

```
foldr(LL(5, -10, 8), lambda x, y: x + 2*y, 0) -> 17
```

Here we have folded the function that adds its first parameter to two times its second parameter over the LList $(1, 2, 3)$ with a base value of 0. The evaluation of this is:

$$5 + 2 * ($$

Using lambda expressions

```
foldr(LL(5, -10, 8), lambda x, y: x + 2*y, 0) -> 17
```

Here we have folded the function that adds its first parameter to two times its second parameter over the `LList` $(1, 2, 3)$ with a base value of 0. The evaluation of this is:

$$
\begin{aligned}
5 + 2 * (& \\
& -10 + 2 * (
\end{aligned}
$$

Using lambda expressions

```
foldr(LL(5, -10, 8), lambda x, y: x + 2*y, 0) -> 17
```

Here we have folded the function that adds its first parameter to two times its second parameter over the `LList` $(1, 2, 3)$ with a base value of 0. The evaluation of this is:

$$
\begin{aligned}
5 + 2 * (& \\
& -10 + 2 * (\\
& \qquad 8 + 2 * 0))
\end{aligned}
$$

Using lambda expressions

```
foldr(LL(5, -10, 8), lambda x, y: x + 2*y, 0) -> 17
```

Here we have folded the function that adds its first parameter to two times its second parameter over the LList $(1, 2, 3)$ with a base value of 0. The evaluation of this is:

$$
\begin{aligned}
&5 + 2 * (\\
&\qquad -10 + 2 * (\\
&\qquad\qquad 8))
\end{aligned}
$$

Using lambda expressions

```
foldr(LL(5, -10, 8), lambda x, y: x + 2*y, 0) -> 17
```

Here we have folded the function that adds its first parameter to two times its second parameter over the `LList` $(1, 2, 3)$ with a base value of 0. The evaluation of this is:

$$
\begin{aligned}
&5 + 2 * (\\
&\qquad\qquad -10 + 2 * 8)
\end{aligned}
$$

Using lambda expressions

```
foldr(LL(5, -10, 8), lambda x, y: x + 2*y, 0) -> 17
```

Here we have folded the function that adds its first parameter to two times its second parameter over the LList $(1, 2, 3)$ with a base value of 0. The evaluation of this is:

$$5 + 2 * (\qquad\qquad 6)$$

Using lambda expressions

```
foldr(LL(5, -10, 8), lambda x, y: x + 2*y, 0) -> 17
```

Here we have folded the function that adds its first parameter to two times its second parameter over the `LList` $(1, 2, 3)$ with a base value of 0. The evaluation of this is:

$$5 + 2 * 6$$

Using lambda expressions

```
foldr(LL(5, -10, 8), lambda x, y: x + 2*y, 0) -> 17
```

Here we have folded the function that adds its first parameter to two times its second parameter over the `LList` $(1, 2, 3)$ with a base value of 0. The evaluation of this is:

$$17$$

Recall — function call expressions

Recall that when we defined our Python function call expressions we said they took the form of

```
expr(argList)
```

Recall — function call expressions

Recall that when we defined our Python function call expressions we said they took the form of

```
expr(argList)
```

When we made this definition we said that `expr` could be any expression that evaluated to a function. Until now the only expressions we had that could evaluate to a function were identifiers. Now, we also have lambda expressions.

Check your understanding

Knowledge Check: Consider the following expressions and state whether they are valid or not. If they are valid state what they will evaluate to, if they are not valid state why they are not.

- `(lambda x, y : x-len(y))(10, "hi")`
- `(lambda x: 3*x)("xyz")`
- `lambda x, y, z : z+x*y`
- `(lambda x : x+3)(2, 5)`

Applications of lambdas

Lambda functions have many applications. One common case is to allow us to quickly write a small simple function to use with a higher-order function like `foldr`.

Applications of lambdas

Lambda functions have many applications. One common case is to allow us to quickly write a small simple function to use with a higher-order function like `foldr`.

Another use case of lambda functions is to allow us to write functions that can return functions!

Our first function returning function

We will now define the function `adderGenerator` which takes a single integer parameter *n*, and returns the function that takes a single integer parameter and adds *n* to it.

```
def adderGenerator(n):
  return lambda x: x + n
```

Our first function returning function

We will now define the function `adderGenerator` which takes a single integer parameter n, and returns the function that takes a single integer parameter and adds n to it.

```
def adderGenerator(n):
  return lambda x: x + n
```

How does this function work? When `adderGenerator` is called the value for n gets bound to the argument that was supplied. As such, in that particular function call of `adderGenerator` n has a particular value and the returned lambda function is the function that adds that particular value to its parameter x.

Our first function returning function

We will now define the function `adderGenerator` which takes a single integer parameter n, and returns the function that takes a single integer parameter and adds n to it.

```
def adderGenerator(n):
  return lambda x: x + n
```

How does this function work? When `adderGenerator` is called the value for n gets bound to the argument that was supplied. As such, in that particular function call of `adderGenerator` n has a particular value and the returned lambda function is the function that adds that particular value to its parameter x.

In this way we say the lambda function *captures* the value of n.

Understanding the `adderGenerator`

The new `adderGenerator` function returns a lambda function with one parameter `x` and whose expression is simply the addition `x + n`. However, when this lambda function is called itself what value is used for `n`?

Understanding the `adderGenerator`

The new `adderGenerator` function returns a lambda function with one parameter `x` and whose expression is simply the addition `x + n`. However, when this lambda function is called itself what value is used for `n`?

In essence, when the lambda function is created the current value of `n` is substituted in place in the function body. In this way each function returned by `adderGenerator` can have a different `n` based on whatever argument was provided to `adderGenerator`.

Understanding the `adderGenerator`

The new `adderGenerator` function returns a lambda function with one parameter `x` and whose expression is simply the addition `x + n`. However, when this lambda function is called itself what value is used for `n`?

In essence, when the lambda function is created the current value of `n` is substituted in place in the function body. In this way each function returned by `adderGenerator` can have a different `n` based on whatever argument was provided to `adderGenerator`.

In functional programming this is called a *closure*. For the simple ways we will use it you may think of it as simply replacing free variables in the function body with their given values at the time of the function's creation.

Sample `adderGenerator` return results

To understand the closures as we are representing them, we demonstrate some function calls of `redShift`, and the resultant values.

`adderGenerator(5)` $\longrightarrow$ `lambda x: x + 5`
`adderGenerator(10)` $\longrightarrow$ `lambda x: x + 10`

In each case, the return value is the lambda function, however the free variable `n` has been replaced with the value it had at the time the lambda function was created, which is during the call to `adderGenerator`.

Simple examples of `adderGenerator`

Now `adderGenerator` is a function that returns a function. We can demonstrate this by playing with it in our interpreter

```
add3 = adderGenerator(3)
add3(0) ⟶ 3
add3(5) ⟶ 8
add10 = adderGenerator(10)
add10(0) ⟶ 10
add10(5) ⟶ 15
```

Example uses of `adderGenerator`

Consider the following applications of `adderGenerator` and their results, we can see the usefulness of the function in composing it with the higher-order function `map`.

Example uses of `adderGenerator`

Consider the following applications of `adderGenerator` and their results, we can see the usefulness of the function in composing it with the higher-order function `map`.

```
map(adderGenerator(3), LL(0, -10, 2)) -> (3, -7, 5)
```

Example uses of `adderGenerator`

Consider the following applications of `adderGenerator` and their results, we can see the usefulness of the function in composing it with the higher-order function `map`.

```
map(adderGenerator(3), LL(0, -10, 2)) -> (3, -7, 5)
map(adderGenerator(5), LL(0, -10, 2)) -> (5, -5, 7)
```

Example uses of `adderGenerator`

Consider the following applications of `adderGenerator` and their results, we can see the usefulness of the function in composing it with the higher-order function `map`.

```
map(adderGenerator(3), LL(0, -10, 2)) -> (3, -7, 5)
map(adderGenerator(5), LL(0, -10, 2)) -> (5, -5, 7)
adderGenerator(10)(2) -> 12
```

Example uses of `adderGenerator`

Consider the following applications of `adderGenerator` and their results, we can see the usefulness of the function in composing it with the higher-order function `map`.

```
map(adderGenerator(3), LL(0, -10, 2)) -> (3, -7, 5)
map(adderGenerator(5), LL(0, -10, 2)) -> (5, -5, 7)
adderGenerator(10)(2) -> 12
```

Note: We now have yet another expression that can appear on the left-hand side of a function call expression in Python — a function call itself! If functions can return functions, then of course a function call can be an expression that evaluates to a function!

Solving the `redShift` problem

Now that we can write functions that return functions, can we solve our `redShift` problem?

Now, `redShift` is the function that returns a unary function which adds the given amount to its arguments red component!

Solving the `redShift` problem

Now that we can write functions that return functions, can we solve our `redShift` problem?

That is, can we write a function `redShift` that takes a parameter *sAmt*, and returns a function that takes a single `Pixel` parameter and adds *sAmt* to its red component, so that we can quickly construct functions to shift by an arbitrary red amount?

Now, `redShift` is the function that returns a unary function which adds the given amount to its arguments red component!

Solving the `redShift` problem

Now that we can write functions that return functions, can we solve our `redShift` problem?

That is, can we write a function `redShift` that takes a parameter *sAmt*, and returns a function that takes a single `Pixel` parameter and adds *sAmt* to its red component, so that we can quickly construct functions to shift by an arbitrary red amount?

Yes we can! Consider the following

Now, `redShift` is the function that returns a unary function which adds the given amount to its arguments red component!

Solving the `redShift` problem

Yes we can! Consider the following

Now, `redShift` is the function that returns a unary function which adds the given amount to its arguments red component!

Solving the `redShift` problem

Yes we can! Consider the following

```
def redShiftHelper(p, sAmt):
  newRed = first(p) + sAmt
  if newRed > 255:
    return cons(255, rest(p))
  return cons(newRed, rest(p))

def redShift(sAmt):
  return lambda p: redShiftHelper(p, sAmt)
```

Now, `redShift` is the function that returns a unary function which adds the given amount to its arguments red component!

Using `redShift`

Now, imagine we have an `LList` of `Pixels` bound to the identifer `img` which is the image we want to transform. We can apply any number of different `redShifts` now by simply mapping different applications of the `redShift` function over our `LList`.

```
map(redShift(30), img)
map(redShift(60), img)
map(redShift(90), img)
```

A simpler `lambda` usage

Consider a simpler use case — there may be a time you want to increment each item of a `LList`. Another time, you may want to add two to each item of a `LList`. Yet another time you may want to add five, etc. This can of course be achieved by mapping unary functions that add one, two, or five respectively to their parameters.

A simpler `lambda` usage

Consider a simpler use case — there may be a time you want to increment each item of a `LList`. Another time, you may want to add two to each item of a `LList`. Yet another time you may want to add five, etc. This can of course be achieved by mapping unary functions that add one, two, or five respectively to their parameters.

We can achieve this by writing the function `adderGenerator`

Lambda limitations

Our lambda functions in Python are somewhat limited due to the fact that their bodies can only be a single expression.

Lambda limitations

Our lambda functions in Python are somewhat limited due to the fact that their bodies can only be a single expression.

This raises the question, could we write the function `redShift` without writing our helper function `redShiftHelper`?

Lambda limitations

Our lambda functions in Python are somewhat limited due to the fact that their bodies can only be a single expression.

This raises the question, could we write the function `redShift` without writing our helper function `redShiftHelper`?

The fact that our `redShiftHelper` function is several statements and expressions would seem to imply that we cannot, but actually...

Lambda limitations

Our lambda functions in Python are somewhat limited due to the fact that their bodies can only be a single expression.

This raises the question, could we write the function `redShift` without writing our helper function `redShiftHelper`?

The fact that our `redShiftHelper` function is several statements and expressions would seem to imply that we cannot, but actually...

We can! We just need another tool in our toolbox, and a clever rewrite of the function.

The ternary operator

Many programming languages have an operator simply named the ternary operator for use in writing conditional expressions.

The ternary operator is used when you want to write an expression that operates on three values a, b, and c, and this expression evaluates to b if a is `True` and c otherwise.

Common uses of the ternary operator

Some of the most common uses of the ternary operator is to simplify code that looks like this

```
if cond:
  x = val1
else:
  x = val2
```

However, since the ternary operator is just that — an operator — it means it is very useful to allow us to write expressions in lambda functions which have conditional evaluation.

Anatomy of the ternary operator

An expression using the ternary operator takes the following form:

```
exprIfTrue if condExpr else exprIfFalse
```

Anatomy of the ternary operator

An expression using the ternary operator takes the following form:

Result if condition is True

```
exprIfTrue if condExpr else exprIfFalse
```

Anatomy of the ternary operator

An expression using the ternary operator takes the following form:

Condition

Result if condition is True

```
exprIfTrue if condAxpr else exprIfFalse
```

Anatomy of the ternary operator

An expression using the ternary operator takes the following form:

Condition

Result if condition is True

```
exprIfTrue if condExpr else exprIfFalse
```

Result if condition is False

Example usage of the ternary operator

```
if cond:
  x = val1
else:
  x = val2
```

So as an example we could rewrite the code above to be instead:

Example usage of the ternary operator

```
if cond:
  x = val1
else:
  x = val2
```

So as an example we could rewrite the code above to be instead:

```
x = val1 if cond else val2
```

Practicing simple lambda functions

Practice Question: Given that you have a `LList` of numbers bound to the identifier *lon* write a single expression that evaluates to the mean of that `LList`. You may not define any named functions, if you require a function for any reason at all it must be a lambda function. You may call the higher-order functions available in the `cmput274` module.

Practicing with lambdas and the ternary

Practice Question: Rewrite the function `redShift` so that it does not require a helper function, and instead simply returns a lambda that produces the correct result.

That is, your lambda should no longer call another helper function as the original one calls `redShiftHelper`.

Inner functions

Due to the restrictions on lambda functions in Python they are sometimes insufficient for writing the return value we would like

Inner functions

Due to the restrictions on lambda functions in Python they are sometimes insufficient for writing the return value we would like

We've seen with `redShift` that one solution to this is to simply write a helper function, and return a lambda that simply calls that helper function, often providing the value for one or more of the parameters.

Inner functions

Due to the restrictions on lambda functions in Python they are sometimes insufficient for writing the return value we would like

We've seen with `redShift` that one solution to this is to simply write a helper function, and return a lambda that simply calls that helper function, often providing the value for one or more of the parameters.

This can also be achieved in Python using *inner functions*. Inner functions are function definitions which take place within another function. As such, their definition is only evaluated when the function they lay within is called — they then "store" the value of the argument used for the outer function's parameter.

Example inner function

```
def redShift(sAmt):
  def redShiftHelper(p):
    newRed = first(p) + sAmt
    if newRed > 255:
      return cons(255, rest(p))
    return cons(newRed, rest(p))
  return redShiftHelper
```

Example inner function

```
def redShift(sAmt):
  def redShiftHelper(p):
    newRed = first(p) + sAmt
    if newRed > 255:
      return cons(255, rest(p))
    return cons(newRed, rest(p))
  return redShiftHelper
```

Here we have moved `redShiftHelper` to be an inner function. Since its definition is inside `redShift` it has access to `sAmt`, which no longer needs to be a parameter of it. As such we can simply return `redShiftHelper`.

Example inner function

```
def redShift(sAmt):
  def redShiftHelper(p):
    newRed = first(p) + sAmt
    if newRed > 255:
      return cons(255, rest(p))
    return cons(newRed, rest(p))
  return redShiftHelper
```

Because `redShiftHelper` is defined each time `redShift` is called, the function itself is also a closure that captures the given value of `sAmt` at the time of its creation.

Inner functions and lambdas

Nothing we can do with inner functions couldn't also be solved by an external helper function and a lambda that calls it, e.g. how we can translate `redShift` between the two implementations.

Inner functions and lambdas

Nothing we can do with inner functions couldn't also be solved by an external helper function and a lambda that calls it, e.g. how we can translate `redShift` between the two implementations.

As such, either lambdas or inner functions can be effective in helping us write functions which return functions.

Practice questions

Practice Question: Write the function `skipGen` which takes a single natural number parameter `n` and returns a function that takes a single `LList` parameter and returns the `LList` that is the result of skipping every `n` elements of it.

```
def skipGen(n):
  '''
  skipGen returns the function f such that when f is applied to
          a LList it returns a new LList the result of skipping
          every nth element of the LList

  n - a natural number > 1
  returns - (LList -> LList)

  Examples:
    skipGen(3)(LL(10,22,36,47,59,60,72,80))
          -> (10, 22, 47, 59, 72, 80)
    skipGen(2)(LL("a","b","c","d"))
          -> ("a", "c")
  '''
```