Multidimensional Data

Rob Hackman

Fall 2025

University of Alberta

Table of Contents

Reconsidering cards

In our previous example of sorting we sought to represent a hand of cards, as for a card game. We did so by representing a card as a `LList` of two items, and a hand as a `LList` of `Cards`.

In this way, a `Hand` was actually represented as a `LList` of fixed size `LLists`. We call such a data type *multidimensional* (specifically in this instance, two-dimensional) as opposed to a `LList` of non-collection data types (such as integers) which we would call one-dimensional.

The dimensions of data

We refer to collections that contain other collections as multidimensional as they are often used to refer to data that is just that. Consider a video game takes place is one dimension. In this game only the horizontal plane exists, so the player may only move left and right without moving up or down.

The dimensions of data

We refer to collections that contain other collections as multidimensional as they are often used to refer to data that is just that. Consider a video game takes place is one dimension. In this game only the horizontal plane exists, so the player may only move left and right without moving up or down.

In this game the player takes up one unit of space which we will call a *tile*. The tiles of our game are one of:

- An open space, represented by the number 0. Players may move into open spaces.
- A wall, represented by the number 1. If a player moves into a wall then they stay in the position they were originally in.

One-Dimensional Warrior

So to implement our game One-Dimensional Warrior we store our world as a tuple of integers representing open spaces or walls. In this way each index of our tuple is a tile in our world.

We also have to represent the position where the player is currently located. Since the indices of our tuple represent the tile in our world we will simply store the integer of the tile where our player is located.

Drawing our world

Since our world is stored as the integers zero and one, and we store our player's location simply as a number, it would be nice to be able to represent our whole world as a string for a nicer way to view it.

Drawing our world

Since our world is stored as the integers zero and one, and we store our player's location simply as a number, it would be nice to be able to represent our whole world as a string for a nicer way to view it.

So we will write the function `gameDisplay` that takes two parameters, the first being the tuple that represents our world and the second being the integer that represents our player's position and returns a string. The length of the string will be the same as the tuple representing our world, and each character in the string will be the character `"|"` if the tile at that position is a wall, the character `"_"` if the tile at that position is open, but if the player is located at that tile it will instead be represented in the string with the character `"T"`.

The gameDisplay function

```
def gameDisplay(world, ppos):
  '''
  gameDisplay returns a string that represents our game of
              One-Dimensional Warrior based on the world
              and player position given.

  world    - A tuple of integers
  ppos      - integer, index of world
  returns  - str

  Examples:
    gameDisplay((1,0,0,0,1), 2) -> "|_T_|"
    gameDisplay((0,0,0,0,0,0,0,0), 0) -> "T_______"
  '''
```

The `gameDisplay` function

This function is a fairly simple `fold` at this point. We simply need to change each tile to the character we want and build a string up from that!

```
def gameDisplay(world, ppos):
  return foldr(world,
               lambda tile, s: tileToChar(tile) + s,
               "")
```

The `gameDisplay` function

But, if we try to write `tileToChar` we find we have a problem...

```
def gameDisplay(world, ppos):
  def tileToChar(t):
    if t == 0:
      return "_"
    elif t == 1:
      return "|"

  return foldr(world,
               lambda tile, s: tileToChar(tile) + s,
               "")
```

The gameDisplay function

Since the function we are folding is operating on the values in the tuple, which represent our tiles, we don't know which index we are at when looking at a given tile.

This means we don't know when the character should be "T"

```
def gameDisplay(world, pos):
  def tileToChar(t):
    if t == 0:
      return "_"
    elif t == 1:
      return "|"

  return foldr(world,
               lambda tile, s: tileToChar(tile) + s,
               "")
```

The gameDisplay function

One way we could solve this problem is to write an explicitly recursive function that recurses over the indices of the tuple instead. For example:

```python
def gameDisplay(world, ppos):
  def gdHelper(curInd):
    if curInd == len(world):
      return ""
    if curInd == ppos:
      return "T" + gdHelper(curInd + 1)
    if world[curInd] == 0:
      return "_" + gdHelper(curInd + 1)
    if world[curInd] == 1:
      return "|" + gdHelper(curInd + 1)

  return gdHelper(0)
```

The `gameDisplay` function

We can achieve this same thing with a fold as well! Instead of folding over the *values* of the tuple we'd like to fold over the *indices* of the tuple. To do so we need to build a sequence of the integers from zero to the length of our tuple minus one.

How can such a sequence be built? `buildList`!

```
def gameDisplay(world, ppos):
```

The gameDisplay function

Now we can write a fold over these indices!

```
def gameDisplay(world, ppos):
  indices = buildList(lambda x: x, len(world))
```

The `gameDisplay` function

And below we have a working function for `gameDisplay`

```
def gameDisplay(world, ppos):
  indices = buildList(lambda x: x, len(world))
  def indToChar(i, s):
    if i == ppos:
      return "T" + s
    # tileToChar written earlier
    return tileToChar(world[i]) + s
  return foldr(indices, indToChar, "")
```

Visualizing one dimensional data

Our silly game, One-Dimensional Warrior, will help us visualize why we are calling a collection of integers one-dimensional and a collection of collections may be called two-dimensional.

Visualizing one dimensional data

Our silly game, One-Dimensional Warrior, will help us visualize why we are calling a collection of integers one-dimensional and a collection of collections may be called two-dimensional.

Now that we know how to display our game as a string let us consider how the indices of our tuple really represent positions in our world, and that we are viewing our tuple as a one-dimensional world

Visualizing one dimensional data

Consider the tuple below that represents one world in the game of One-Dimensional Warrior. Underneath the values of the tuple are the indices of each of those values.

```
(1, 0, 0, 0, 0, 0, 1)
 0, 1, 2, 3, 4, 5, 6
```

Now if our player is at position or *coordinate* one in our world, what does that tell us about what that player can do?

Visualizing one dimensional data

Consider the tuple below that represents one world in the game of One-Dimensional Warrior. Underneath the values of the tuple are the indices of each of those values.

```
(1, 0, 0, 0, 0, 0, 1)
 0, 1, 2, 3, 4, 5, 6
```

Now if our player is at position or *coordinate* one in our world, what does that tell us about what that player can do?

We have to agree on some mapping between our data representation (a sequential tuple of integers representing tiles of our world) and our world itself.

Visualizing one dimensional data

```
(1, 0, 0, 0, 0, 0, 1)
 0, 1, 2, 3, 4, 5, 6
```

One mapping we could choose is that the indices of our tuple represent x-coordinates in a one-dimensional Euclidean space. Our x-coordinates will represent movement in our plane from left to right with smaller numbers being left of larger numbers.

Visualizing one dimensional data

```
(1, 0, 0, 0, 0, 0, 1)
 0, 1, 2, 3, 4, 5, 6
```

One mapping we could choose is that the indices of our tuple represent x-coordinates in a one-dimensional Euclidean space. Our x-coordinates will represent movement in our plane from left to right with smaller numbers being left of larger numbers.

That means our player is located at coordinate one, then they could not move to the left as there is a wall at coordinate zero but they could move to the right as coordinate two is an open tile.

Movement in One-Dimensional Warror

Practice Question: Write a function `validMoves` that takes a tuple that represents our world in One-Dimensional Warrior and an integer that represents the players current position.

Your function should return a `LList` of strings that contains zero, one, or both of the strings `"left"` or `"right"` if the character is able to move left, or right, both, or neither. Examples:

```
validMoves((1, 0, 0, 0, 0, 0, 1), 1) -> ("right")
validMoves((1, 0, 0, 0, 0, 0, 1), 3) -> ("left", "right")
validMoves((1, 0, 1), 1) -> ()
validMoves((0, 0, 0, 0), 3) -> ("left")
```

validMoves

```python
def validMoves(world, ppos):
  leftPos = ppos - 1
  dirs = ()
  if leftPos >= 0 and world[leftPos] == 0:
    dirs = dirs + ("left",)
  rightPos = ppos + 1
  if rightPos < len(world) and world[rightPos] == 0:
    dirs = dirs + ("right",)
  return dirs
```

Mapping indices to coordinates

In order to write the function `validMoves` we had to understand some details about our mapping between the indices of the tuple that represents our game world and the Euclidean plane our game takes place in.

- Our world is finite, and anything to the left of index zero does not exist, and anything to the right of the largest index in our tuple does not exist
- The position directly left of the position with x-coordinate ω is the position with x-coordinate $\omega - 1$
- The position directly right of the position with x-coordinate ω is the position with x-coordinate $\omega + 1$
- Thus, decrementing an x-coordinate is the way to move left, and incrementing an x-coordinate is the way to move right.

Adding enemies

We will now add enemies to our game. As we may have many enemies we will represent the enemies with a tuple of the coordinates of all enemies. So now our game state has three pieces of data:

- `world` — a tuple of integers, representing open spaces or walls with zeroes and operations
- `enemies` — a tuple of integers, representing x-coordinates in our world where enemies reside
- `ppos` — a single integer, representing the x-coordinate of our player

We will update our `gameDisplay` to representing positions with enemies as the character `"X"`

Updated gameDisplay

```
def gameDisplay(world, enemies, ppos):
  indices = buildList(lambda x: x, len(world))
  def indToChar(i, s):
    if i == ppos:
      return "T" + s
    if foldr(enemies, lambda x, res: x == i or res, False):
      return "X" + s
    return tileToChar(world[i]) + s
  return foldr(indices, indToChar, "")

'''
Example:
  gameDisplay((0,0,0,0,0,0), (1, 4), 2) -> '_XT_X_'
'''
```

Enemy interaction

If our player character touches an enemy then our player character dies and loses the game.

Enemy interaction

If our player character touches an enemy then our player character dies and loses the game.

This new introduction of enemies introduces a flaw in our game design — currently the player has no way to avoid an enemy in their way!

Enemy interaction

If our player character touches an enemy then our player character dies and loses the game.

This new introduction of enemies introduces a flaw in our game design — currently the player has no way to avoid an enemy in their way!

We may decide that we would like our players to be able to fly over the enemies. To fly, however, would be to move vertically — a dimension that does not currently exist in our game!

A new dimension

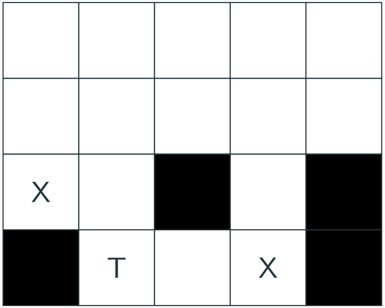
If we want to allow our player character to move in the vertical axis a well as the horizontal axis of our Euclidean plane we must extend our world to contain tiles in both directions.

Adding a new dimension also means coming up with a method of storing the relationships between our tiles. For any give two tiles T_a and T_b we must know where T_a is relative to T_b and vice versa.

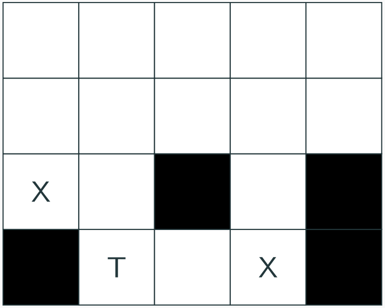
Previously we used indices to represent coordinates in the horizontal axis, we will do the same again for our new dimension.

Representing a two-dimensional world

Consider the following diagram that represents the world in our new game *Two-Dimensional Warrior*. A white box represents an open space, while a black box represents a wall, an `X` in a box represents an enemy, and a `T` in a box represents our player character.



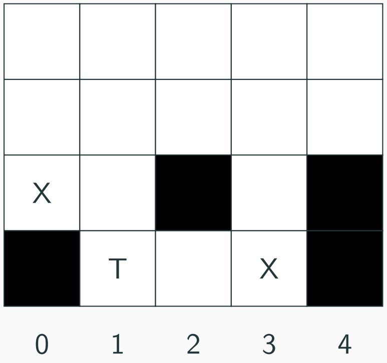
Representing a two-dimensional world



It becomes clear in our two-dimensional world we just have a series of our one-dimensional worlds stacked on top of each other.

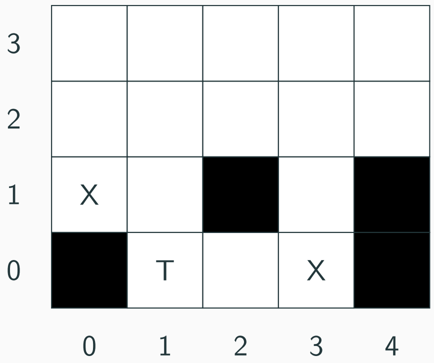
As such, we can choose to represent this data in our program as a tuple that stores the horizontal tiles of our world, which we can think of as the *rows* of our world.

Representing a two-dimensional world



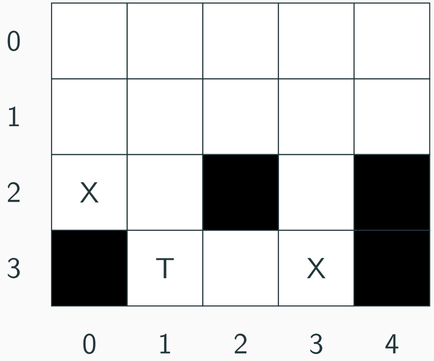
Above we now have our x-axis labelled with the index values, as it would have been when we were only in a one-dimensional world.

Representing a two-dimensional world



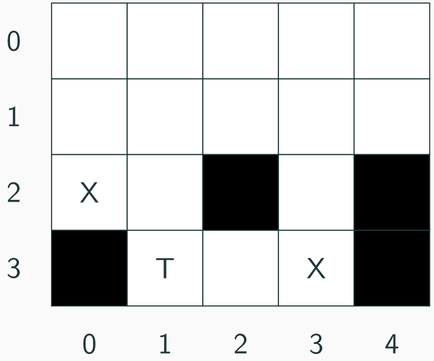
Now we want a similar relationship between integers and our y-axis. The typical mapping of integers and positions on the y-axis is that as integers get larger one is higher in the y-axis, and 0 is the bottom of the axis.

Representing a two-dimensional world



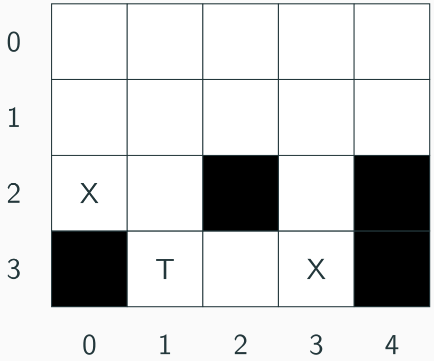
Despite this mapping being common we will instead choose a mapping where the *top* of the y-axis is position 0, and as y-coordinates get larger we are actually going further down in the y-axis.

Representing a two-dimensional world



Knowledge Check: Given this representation of our world we must now update the positions of our player and enemies to be pairs of integers, both an `x` and `y` coordinate. If we represent the coordinate of an entity as a tuple of the form `(x,y)` what literal would represent our `ppos` variable? What literal would represent our `enemies` variable?

Representing a two-dimensional world



Knowledge Check:

```
ppos = (1, 3)
enemies = ((0, 2), (3, 3))
```

Representing our two-dimensional world

As we've observed our two-dimensional world is really just several of our previous one-dimensional worlds. As such, we can implement our two-dimensional world as a tuple that contains tuples of integers which are the rows of our world.

```
world = ((0, 0, 0, 0, 0),
         (0, 0, 0, 0, 0),
         (0, 0, 1, 0, 1),
         (1, 0, 0, 0, 1))
```

For example the tuple `world` above stored four tuples, just as our world in the previous slide had four rows. Each row stores the integers zero or one to represent if a wall is at that position or not.

Accessing a two-dimensional data type

Given the following definition:

```
twoDim = (("a", "b", "c"), ("d", "e", "f"), ("g", "h", "i"))
```

What is a single Python expression that would evaluate to the string `"d"` using only the identifier `twoDim` and indexing?

Accessing a two-dimensional data type

Given the following definition:

```
twoDim = (("a", "b", "c"), ("d", "e", "f"), ("g", "h", "i"))
```

What is a single Python expression that would evaluate to the string `"d"` using only the identifier `twoDim` and indexing?

```
twoDim[1][0]
```

Since `twoDim[1]` is an expression that evaluates to the second item in the sequence `twoDim` it evaluates to the tuple `("d", "e", "f")`. Then we are left with the expression `("d", "e", "f")[0]` which evaluates to the first item of that sequence which is `"d"`.

Accessing a two-dimensional data type

```
twoDim = (("a", "b", "c"),
          ("d", "e", "f"),
          ("g", "h", "i"))
twoDim[1][0] -> "d"
twoDim[2][1] -> "h"
```

Note: When representing data in two dimensions as a sequence of *rows* then the first index is to select the row and the index into the row is to select the individual value.

That means for our `tuple` that represents the world in our Two-Dimensional Warrior game we must index our `world` with a y-coordinate to access that row and then access that row with a given x-coordinate to get the value at that coordinate.

Updating `validMoves`

Now that we have added an entire new dimension to our games world we must updated `validMoves` to support the ability that the player may fly up or down as well as moving to the left or the right.

So we need to update `validMoves` so it also considers the space above the player and the space below the player and may also include in the returned options the strings `"up"` or `"down"`.

Updated `validMoves`

First we must acknowledge a change in types of `world` and `ppos`

where `world` used to be a `tuple of ints` and `ppos` used to be an integer now `world` is a `tuple of tuple of int` and `ppos` is a `tuple(int, int)`.

So while the position to the left of `ppos` was previously `ppos-1` in One-Dimensional Warrior now if the player is at the position (x, y) then the position to the left of them is the position $(x - 1, y)$

```
def validMoves(world, ppos):
  dirs = ()
  leftPos = (ppos[0] - 1, ppos[1])
```

Updated `validMoves`

Similarly to before, the position to our left is valid only if it is within our world (the new x-coordinate is 0 or greater) and it is not a wall. However, checking for these conditions is slightly different.

```
def validMoves(world, ppos):
  dirs = ()
  leftPos = (ppos[0] - 1, ppos[1])
  if leftPos[0] >= 0 and world[leftPos[1]][leftPos[0]] == 0:
    dirs = dirs + ("left",)
```

Note: When wanting to know whether the tile at the coordinate `leftPos` is a wall or not we must index our `world` with `leftPos[1]` first, as that indicates the y-coordinate and our `world` is stored as a tuple of horizontal rows.

Updated `validMoves`

We must now check if the position to our right is valid and open, to determine if we can move to the right.

```
def validMoves(world, ppos):
  ... # All code shown before
  rightPos = (ppos[0] + 1, ppos[1])
  if rightPos[0] < len(world[ppos[1]]) and \
     world[rightPos[1]][rightPos[0]]:
    dirs = dirs + ("right",)
```

Note: We can no longer use `len(world)` to tell us if the position to our right is within the bounds of our world! The length of `world` tells us how many rows total we have, which is to say the vertical height of our world! The horizontal width of our world is determined by the length of the rows stored in `world`.

Updated `validMoves`

Now we must check above and below us to determine if we can fly up or down. While moving to the left or right was decrementing or incrementing our x-coordinate respectively, moving up or down is decrementing or incrementing our y-coordinate respectively.

```
def validMoves(world, ppos):
  ... # All code shown before
  upPos = (ppos[0], ppos[1] - 1)
  if upPos[1] >= 0 and world[upPos[1]][upPos[0]]:
    dirs = dirs + ("up",)
```

Note: Due to the nature of how we've represented our world in our program moving up is *decrementing* the y-coordinate, while moving down is *incrementing* the y-coordinate.

Updated `validMoves`

Lastly we must check if we can move down.

```
def validMoves(world, ppos):
  ... # All code shown before
  downPos = (ppos[0], ppos[1] + 1)
  if downPos[1] < len(world) and \
     world[downPos[1]][downPos[0]]:
    dirs = dirs + ("up",)
  return dirs
```

Note: When checking if `upPos` has gone too far down we must check if its y-coordinate is still a valid row index. In this case we *should* use `len(world)` since as `world` is a tuple of rows, the length of `world` is how many rows we have!

Updated `validMoves`

```
def validMoves(world, ppos):
  dirs = ()
  leftPos = (ppos[0] - 1, ppos[1])
  if leftPos[0] >= 0 and world[leftPos[1]][leftPos[0]] == 0:
    dirs = dirs + ("left",)
  rightPos = (ppos[0] + 1, ppos[1])
  if rightPos[0] < len(world[ppos[1]]) and \
     world[rightPos[1]][rightPos[0]]:
    dirs = dirs + ("right",)
  upPos = (ppos[0], ppos[1] - 1)
  if upPos[1] >= 0 and world[upPos[1]][upPos[0]]:
    dirs = dirs + ("up",)
  downPos = (ppos[0], ppos[1] + 1)
  if downPos[1] < len(world) and \
     world[downPos[1]][downPos[0]]:
    dirs = dirs + ("up",)
  return dirs
```

Updated `validMoves`

However, a lot of the code here is very similar! In fact it's all doing the exact same abstract procedure — determine a new position, check if that position is valid and open, and if it is then append a new string to our tuple.

Perhaps this code could be simplified if we simply wrote a helper function that takes three parameters, `world`, `pos`, and `dirString`. If the position `pos` is both a valid and open position in `world` then the function returns a tuple that contains simply the string `dirString`. If `pos` is either not valid or not open then the function returns simply the empty tuple.

Updated `validMoves`

We started by writing a helper function `validAndOpen` that returns True if a given position is both a valid position in our world and corresponds to an open tile, and False otherwise.

```
def validAndOpen(world, pos):
  # As always pos is a coordinate of the form (x, y)
  x = pos[0]
  y = pos[1]
  # Knowledge check:
  # this function doesn't work anymore if the order of
  # these if statements is swapped. Why not?
  if y < 0 or y >= len(world):
    return False
  if x < 0 or x >= len(world[y]):
    return False
  return world[y][x] == 0
```

Updated validMoves

Now consider the following rewrite of validMoves.

```
def validMoves(world, ppos):
  dirTupleOrNot = lambda p, dirS: (dirS,) if validAndOpen(world, p) \
                                          else ()
  dirs = ()
  leftPos = (ppos[0] - 1, ppos[1])
  dirs = dirs + dirTupleOrNot(leftPos, "left")
  rightPos = (ppos[0] + 1, ppos[1])
  dirs = dirs + dirTupleOrNot(rightPos, "right")
  upPos = (ppos[0], ppos[1] - 1)
  dirs = dirs + dirTupleOrNot(upPos, "up")
  downPos = (ppos[0], ppos[1] + 1)
  dirs = dirs + dirTupleOrNot(downPos, "down")
  return dirs
```

Still too much repetition!

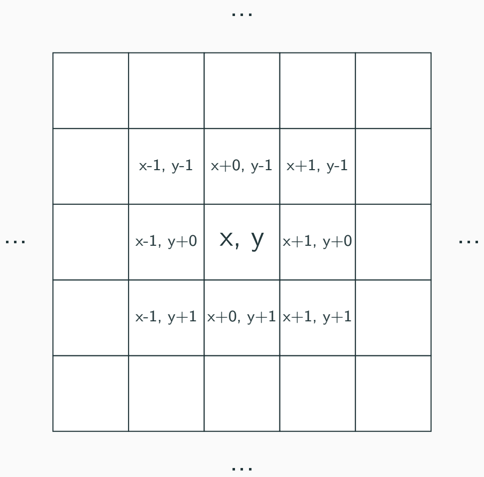
Updated `validMoves`

We want to repeat a process for each pair of position and named direction, so we could rewrite this further using multi-dimensional data!

```
def validMoves(world, ppos):
  dirTupleOrNot = lambda p, dirS: (dirS,) if validAndOpen(world, p) \
                                          else ()
  posDirPairs = ( ((ppos[0] - 1, ppos[1]), "left"),
                  ((ppos[0] + 1, ppos[1]), "right"),
                  ((ppos[0], ppos[1] - 1), "up"),
                  ((ppos[0], ppos[1] + 1), "down"))
  foldPairs = lambda p, dirs: dirTupleOrNot(p[0], p[1]) + dirs
  return foldr(posDirPairs, foldPairs, ())
```

Visualizing coordinates of two-dimensional data

We now consider given an arbitrary two-dimensional sequence and the coordinates (x, y) what are other square coordinates relatively?



Visualizing coordinates of two-dimensional data

Knowledge Check: Relative to our known coordinate (x, y) what is the coordinate of the tile with an A in it? What about the tile with a B? What aoubt the tile with a C?

A				
	x-1, y-1	x+0, y-1	x+1, y-1	
	x-1, y+0	x, y	x+1, y+0	C
	x-1, y+1	x+0, y+1	x+1, y+1	
	B			

Practice Question

In the game MineSweeper players click on an $n \times m$ grid trying not to click on mines. Each tile in the grid is either a mine or an open space.

Given the following data definition for the data type `MSGrid` write the function `adjacentBombs` that takes a `MSGrid` and two integers representing an x and y coordinate and returns the number of adjacent bombs to the given position.

```
A MSGrid is a tuple of tuples of MSChar
```

```
A MSChar is one of "B" or "O"
to represent a bomb or open space respectively
```

Examples for `adjacentBombs`

```
# Note, when definining a multidimensional data type
# we do not need to separate the rows with newlines,
# doing so is simply for readability.
# The grid defined as g below would be the
# same if we defined it simply as:
# g = (("B", "O", "B"), ("O", "B", "O"), ("B", "B", "B"))
g = (("B", "O", "B"),
     ("O", "B", "O"),
     ("B", "B", "B"))
adjacentBombs(g, 0, 1) -> 4
adjacentBombs(g, 1, 1) -> 5
```

Building two-dimensional data

So far we have operated on existing two-dimensional data, which we have provided in the form of a literal value.

We must often write functions that produce two-dimensional data, rather than assume it is given.

For example, if we were making a card game we may want to write a function that produces an entire deck of cards, stored in *new deck order*[1].

[1] New deck order is a specific ordering of cards. First are the jokers, then for each suit the cards will be in order Ace to King with the suits appearing in the order Spades, Diamonds, Clubs, Hearts. We also have no jokers in our cards.

Building a new deck order deck of cards

A deck of cards can simply be represented as a `LList` of our `Card` data type. Since our `Cards` are a `LList` of two integers then our `Deck` is a `LList of LL(int, int)` which is a two-dimensional data type.

Then to build our `Deck` of cards we must repeat to produce every single card. That means for any one suit creating one card for each rank, and repeating that process for each suit.

Building all cards of a suit

First, let's start with writing the function that allows us to build each card of a suit. Let's call this function `allRanks` which takes a number that represents one of our suits and returns a `LList` of all the cards in that suit in the order Ace to King.

```
def allRanks(suit):
```

Building all cards of a suit

As our ranks are the integers in the range $[2, 14]$ we must repeatedly create a card once for each rank. So we will write the function that takes two integer *n* and *cur* and returns all the cards of this suit from *cur* to *n*.

```python
def allRanks(suit):
  def ranksUpTo(n, cur):
    if n == cur:
      return cons(LL(n, suit), empty())
    return cons(LL(cur, suit), ranksUpTo(cur+1, n))
```

Building all cards of a suit

This function allows us to build a LList of all the ranks we want. Except, if we simply call it as ranksUpTo(2, 14) then we would get the LList that has the Ace at the end, not at the start as we want. This can be solved simply by producing the LList of cards from the two until the King and simply prepending the Ace.

```
def allRanks(suit):
  def ranksUpTo(cur, n):
    if n == cur:
      return cons(LL(n, suit), empty())
    return cons(LL(cur, suit), ranksUpTo(cur+1, n))
  return cons(LL(14, suit), ranksUpTo(2, 13))
```

Building all cards of all suits

Now that we have the function `allRanks` we can write our function `newDeck` that returns a new deck of cards in new deck order. Since `newDeck` always returns the same thing, it has no parameters and is a constant function. We start by creating a `LList` of the suits in the order we want to produce them.

```
def newDeck():
  suitOrder = LL(3, 0, 1, 2)
```

Building all cards of all suits

Now for each suit we have we must call `allRanks` which will give us a `LList` of all the ranks of that suit. We then simply must combine all of these `LLists` into one `LList` to finish our function. We could start by mapping `allRanks` onto our `LList` of suits.

```
def newDeck():
  suitOrder = LL(3, 0, 1, 2)
  listOfSuits = map(allRanks, suitOrder)
```

Building all cards of all suits

Now `listOfSuits` is a `LList` of four `LLists`, each of the four `LLists` is the `LList` of all the cards of each suit. We want to combine each of these `LLists` into one `LList` of cards, so we want to concatenate all of these `LLists`.

```
def newDeck():
  suitOrder = LL(3, 0, 1, 2)
  listOfSuits = map(allRanks, suitOrder)
  concatenate = lambda l1, l2: foldr(l1, cons, l2)
  return foldr(listOfSuits, concatenate, empty())
```

Multiplication Tables

Practice Question: Write the function `multTables` that takes two parameters `numRows` and `numCols` and returns a `LList` of `numRows` `LLists` of `numCols` integers.

The `LList` you produce should represent the $numRows \times numCols$ multiplication table, which is that any entry at row i and column j should be the value $i \times j$. We will refer to columns and row numbers as starting at one not zero for the purpose of defining the multiplication table, but of course our real indices start at 0.

Examples of `multTables`

```
multTables(3,5)
  -> LL(LL(1, 2, 3, 4, 5),
        LL(2, 4, 6, 8, 10),
        LL(3, 6, 9, 12, 15))

multTables(4,2)
  -> LL(LL(1, 2),
        LL(2, 4),
        LL(3, 6),
        LL(4, 8))
```

Table of Contents

Arbitrary sequences

So far we have worked with very simplistic multidimensional data. Namely, we have always had the two following assumptions:

We now will start to consider examples where this is not always the case.

Arbitrary sequences

So far we have worked with very simplistic multidimensional data. Namely, we have always had the two following assumptions:

- Our sequence is only two-dimensions

We now will start to consider examples where this is not always the case.

Arbitrary sequences

So far we have worked with very simplistic multidimensional data. Namely, we have always had the two following assumptions:

- Our sequence is only two-dimensions
- The dimensions are fixed-size, that is we always have a perfect rectangular grid of $n \times m$ items

We now will start to consider examples where this is not always the case.

Calculating wages

Consider you are running a business and you have several employees. As such you have a `LList` of employees and the hours they worked. Each day when an employee works a shift you add the hours they worked to their `LList` and come each pay day you use this `LList` to calculate the amount you owe each employee.

As such your `PayrollList` is of the form

```
A PayrollList is a:
 - cons(str, LList of int)
```

So it is a `LList` in which the first item is a string representing the name of the employee and the rest of the `LList` is all integers, which are the number of hours they worked.

`wagesDue` **function**

We will now write the function `wagesDue` that takes a `PayRollList` and returns a `LList` of pairs. Each pair is of the form `LL(str, float)` where the first item is an employee's name and the second item is the amount of wages due to that employee this pay period.

Each employee receives the same wage of 22.50 per hour worked.

The `wagesDue` function

```
def wagesDue(pList):
  '''
  wagesDue takes a PayRollList and returns a LList of pairs
          of all the employees and how much money they are
          owed for this pay period.

  pList   - PayRollList
  returns - LList of LL(str, float)

  Example:
    wagesDue(LL(LL("Erik", 5, 8, 4, 7),
                LL("Ashley", 6, 2),
                LL("Stefan", 2, 2, 2, 2, 2, 2, 2))
      -> LL(LL("Erik", 540.0),
            LL("Ashley", 180.0),
            LL("Stefan", 315.0))
  '''
```

The wagesDue function

For each LList in our PayRollList we want to construct a pair. As such, we want to find a function we can map onto our PayRollList to produce each pair.

```
def wagesDue(pList):
  return map(???, pList)
```

The `wagesDue` function

In this case we need a function that takes all items in each `LList` except for the first one, sums them up, and multiplies them by 22.50. That can be achieved very easily with a `fold`!

```
def wagesDue(pList):
  def hoursToPair(hList):
    totalHrs = foldr(rest(hList), lambda hr, tot: hr + total, 0)
    return LL(first(hList), 22.50*totalHrs)
  return map(hoursToPair, pList)
```

The wagesDue function

The problem of working with multi-dimensional data can often be broken down into subproblems. In simply had to solve the problem for *one* LList and could map that solution onto our LLists of LLists. In this way, thinking recursively and with higher order functions can be very beneficial to working with multidimensional data.

```
def wagesDue(pList):
  def hoursToPair(hList):
    totalHrs = foldr(rest(hList), lambda hr, tot: hr + total, 0)
    return LL(first(hList), 22.50*totalHrs)
  return map(hoursToPair, pList)
```

Working with arbitrary nested collections

When a collection data type is contained within another, such as a `LList` within a `LList` we say that the inner `LList` is *nested* within the outer `LList`.

Working with arbitrary nested collections

When a collection data type is contained within another, such as a `LList` within a `LList` we say that the inner `LList` is *nested* within the outer `LList`.

So far we have only worked with collections that contained collections of types like integers. However, our `LList` and `tuple` sequences can store *anything*. Therefore, a `LList` could store a `LList` that itself stores `LLists`... This nesting could go to an arbitrary depth!

Working with arbitrary nested collections

When a collection data type is contained within another, such as a `LList` within a `LList` we say that the inner `LList` is *nested* within the outer `LList`.

So far we have only worked with collections that contained collections of types like integers. However, our `LList` and `tuple` sequences can store *anything*. Therefore, a `LList` could store a `LList` that itself stores `LLists`... This nesting could go to an arbitrary depth!

Truly arbitrary data is rarely useful (data that contains absolutely anything). However, data of arbitrary depth can certainly come in handy and as such we must learn to work with it.

A SuperLList

Since a LList can already contain anything it can already be arbitrarily nested. However, to give some understanding to the data we are working with we are going to define a SuperLList as follows:

```
A SuperLList of X is one of:
 - empty()
 - cons(X, SuperLList of X)
 - cons(SuperLList of X, SuperLList of X)
```

In this way we can define a SuperLList of any data type and that LList will only contain the given data type or LLists. The LLists it contains must also only contain the given data type or LLists that follow the same rules.

Understanding `SuperLLists`

So if we wanted to define some `SuperLLists of ints` we could begin with the base case of our recursive definition.

Understanding `SuperLLists`

So if we wanted to define some `SuperLLists of ints` we could begin with the base case of our recursive definition.

- `empty()` is a `SuperLList` of ints, by the definition.

Understanding `SuperLLists`

So if we wanted to define some `SuperLLists of ints` we could begin with the base case of our recursive definition.

- `empty()` is a `SuperLList` of ints, by the definition.
- `cons(1, empty())` is also a `SuperLList` of ints, since 1 is an int and `empty()` is a `SuperLList` of ints.

Understanding `SuperLLists`

So if we wanted to define some `SuperLLists of ints` we could begin with the base case of our recursive definition.

- `empty()` is a `SuperLList` of ints, by the definition.
- `cons(1, empty())` is also a `SuperLList` of ints, since 1 is an int and `empty()` is a `SuperLList` of ints.
- `cons(cons(1, empty()), cons(1, empty()))` then is also a `SuperLList` of ints since both arguments to `cons` are `SuperLList` of ints.

Understanding `SuperLLists`

So if we wanted to define some `SuperLLists of ints` we could begin with the base case of our recursive definition.

- `empty()` is a `SuperLList` of ints, by the definition.
- `cons(1, empty())` is also a `SuperLList` of ints, since 1 is an int and `empty()` is a `SuperLList` of ints.
- `cons(cons(1, empty()), cons(1, empty()))` then is also a `SuperLList` of ints since both arguments to cons are `SuperLList` of ints.
- Note that the expression above is the `LList ((1), 1)` and *not* the `LList (1, 1)`

Understanding `SuperLLists`

So if we wanted to define some `SuperLLists of ints` we could begin with the base case of our recursive definition.
Then the follow expression is *also* `SuperLLists` of ints.

```
cons(LL(LL(1), 1), LL(LL(1), 1)) -> (((1), 1), (1), 1)
```

Note that this is a `LList` of three items, the first is the `LList` `((1), 1)`, the second is the `LList` `(1)`, and the third is the integer `1`

From these rules we can build any `LList` that contains only other `LLists` and integers, and the same is true for each of the `LLists` contained within.

Working with a `SuperLList`

So now we would like to write a function that operates on a `SuperLList` of integers. We will try to write the function `sumSuperList` that takes a `SuperLList` of ints and returns the sum of *all* integers that are contained within the `SuperLList`, no matter how deeply nested that integer is.

To solve this problem we must not forget all we have learned, and think towards our data definition and what the possibilities we have to deal with are, and when we know we are finished.

Writing `sumSuperList`

To solve our problem we think must think back to our data definition of our `SuperLList` and determine what we know, and when our problem is trivially solvable.

```
def sumSuperList(sl):
  '''
  sumSuperList return the sum of all integers in the
              SuperLList

  sl      - SuperLList of ints
  returns - int

  Example:
    sumSuperList(LL(5, 3,
                    LL(2, LL(1, 4), LL(6)),
                    LL(LL(LL(LL(1)))))) -> 22
  '''
```

Writing `sumSuperList`

Our data definition makes it clear we have three cases when looking at any given `SuperLList` of ints.

So, we must determine how to handle each case in our function!

```
'''
A SuperLList of ints is one of
 - empty()
 - cons(int, SuperLList of ints)
 - cons(SuperLList of ints, SuperLList of ints)
'''
```

Writing `sumSuperList`

Our data definition makes it clear we have three cases when looking at any given `SuperLList` of ints.

- The `SuperLList` is empty

So, we must determine how to handle each case in our function!

```
'''
A SuperLList of ints is one of
 - empty()
 - cons(int, SuperLList of ints)
 - cons(SuperLList of ints, SuperLList of ints)
'''
```

Writing `sumSuperList`

Our data definition makes it clear we have three cases when looking at any given `SuperLList` of ints.

- The `SuperLList` is empty
- The `SuperLList` is not empty and the first item is an integer

So, we must determine how to handle each case in our function!

```
'''
A SuperLList of ints is one of
 - empty()
 - cons(int, SuperLList of ints)
 - cons(SuperLList of ints, SuperLList of ints)
'''
```

Writing `sumSuperList`

Our data definition makes it clear we have three cases when looking at any given `SuperLList` of ints.

- The `SuperLList` is empty
- The `SuperLList` is not empty and the first item is an integer
- The `SuperLList` is not empty and the first item is a `SuperLList`

So, we must determine how to handle each case in our function!

```
'''
A SuperLList of ints is one of
 - empty()
 - cons(int, SuperLList of ints)
 - cons(SuperLList of ints, SuperLList of ints)
'''
```

Writing `sumSuperList`

Our data definition makes it clear we have three cases when looking at any given `SuperLList` of ints.

- The `SuperLList` is empty
- The `SuperLList` is not empty and the first item is an integer
- The `SuperLList` is not empty and the first item is a `SuperLList`

So, we must determine how to handle each case in our function!

```
'''
A SuperLList of ints is one of
 - empty()
 - cons(int, SuperLList of ints)
 - cons(SuperLList of ints, SuperLList of ints)
'''
```

Writing sumSuperList

The first case, when the SuperLList is empty, is easy. The sum of an empty SuperLList of integers is of course zero.

```
def sumSuperList(sl):
  if isEmpty(sl):
    return 0
```

Writing sumSuperList

The first case, when the SuperLList is empty, is easy. The sum of an empty SuperLList of integers is of course zero.

But how do we handle the other cases? Actually, before we even decide what to do in each case how can we even check which case we have?

```
def sumSuperList(sl):
  if isEmpty(sl):
    return 0
```

Writing sumSuperList

The first case, when the SuperLList is empty, is easy. The sum of an empty SuperLList of integers is of course zero.

But how do we handle the other cases? Actually, before we even decide what to do in each case how can we even check which case we have?

We must first learn of a new function, and concept, in Python.

```
def sumSuperList(sl):
  if isEmpty(sl):
    return 0
```

The `type` function

The `type` function is a built-in function that takes a single parameter of any type and returns the *type* of the given argument.

The `type` function

The `type` function is a built-in function that takes a single parameter of any type and returns the *type* of the given argument.

Since the function `type` returns the type of a piece of data, it means that types themselves must be values in Python (since otherwise how could they be returned?).

The `type` function

The `type` function is a built-in function that takes a single parameter of any type and returns the *type* of the given argument.

Since the function `type` returns the type of a piece of data, it means that types themselves must be values in Python (since otherwise how could they be returned?).

Types themselves can also be used as values in Python, as with functions. If you want to refer to the value that represents a type you can do so with the identifier that is the name of the type.

Using the `type` function

Below are some expressions in Python followed by what they display if evaluated in the Python interpreter.

```
int -> <class 'int'>
float -> <class 'float'>
type(5) -> <class 'int'>
type("hello") -> <class 'str'>
type(3) == int -> True
type(7) == float -> False
```

Note that `<class ...>` is simply how the Python interpreter displays a type, so the values being used here are types. We see in our last two expressions that types can be compared!

Finishing `superSumList`

Now that we know about the `type` function, and the fact that we can compare types, we can finish our function. We know if our `SuperLList` is not empty then the first item is either an `int` or another `SuperLList` so let us begin by checking if it is an `int`.

```
def sumSuperList(sl):
  if isEmpty(sl):
    return 0
  if type(first(sl)) == int:
```

Finishing `superSumList`

If the first item is an `int` then we need only to add it to the sum of the rest of this `SuperLList`.

```
def sumSuperList(sl):
  if isEmpty(sl):
    return 0
  if type(first(sl)) == int:
    return first(sl) + sumSuperList(rest(sl))
```

Finishing `superSumList`

Now if the first item is not an `int` it must be a `LList`. Still, included here is how to check if it is in fact a `LList`

```
def sumSuperList(sl):
  if isEmpty(sl):
    return 0
  if type(first(sl)) == int:
    return first(sl) + sumSuperList(rest(sl))
  if type(first(sl)) == LList:
```

Finishing `superSumList`

If the first item is a `SuperLList` then we need to sum all the integers within in, regardless of how nested they are, and add that to the sum of the rest of our `SuperLList` parameter.

Well the function `sumSuperList` does just that!

```
def sumSuperList(sl):
  if isEmpty(sl):
    return 0
  if type(first(sl)) == int:
    return first(sl) + sumSuperList(rest(sl))
  if type(first(sl)) == LList:
    return sumSuperList(first(sl)) + sumSuperList(rest(sl))
```

Finishing superSumList

And now, using the power of recursion, our sumSuperList function is complete!
Well the function sumSuperList does just that!

```
def sumSuperList(sl):
  if isEmpty(sl):
    return 0
  if type(first(sl)) == int:
    return first(sl) + sumSuperList(rest(sl))
  if type(first(sl)) == LList:
    return sumSuperList(first(sl)) + sumSuperList(rest(sl))
```

Tracing `sumSuperList`

In order to understand how `sumSuperList` (which we shorten to `ssl`) works, we trace out one application of it:

Tracing `sumSuperList`

In order to understand how `sumSuperList` (which we shorten to `ssl`) works, we trace out one application of it:

```
ssl((5, 3, (2, (1, 4, (6))), 9))
```

Tracing `sumSuperList`

In order to understand how `sumSuperList` (which we shorten to `ssl`) works, we trace out one application of it:

```
ssl((5, 3, (2, (1, 4, (6))), 9))
5 + ssl((3, (2, (1, 4, (6))), 9))
```

Tracing `sumSuperList`

In order to understand how `sumSuperList` (which we shorten to `ssl`) works, we trace out one application of it:

```
ssl((5, 3, (2, (1, 4, (6))), 9))
5 + ssl((3, (2, (1, 4, (6))), 9))
5 + 3 + ssl(((2, (1, 4, (6))), 9))
```

Tracing `sumSuperList`

In order to understand how `sumSuperList` (which we shorten to `ssl`) works, we trace out one application of it:

```
ssl((5, 3, (2, (1, 4, (6))), 9))
5 + ssl((3, (2, (1, 4, (6))), 9))
5 + 3 + ssl(((2, (1, 4, (6))), 9))
5 + 3 + ssl((2, (1, 4, (6)))) + ssl((9))
```

Tracing `sumSuperList`

In order to understand how `sumSuperList` (which we shorten to `ssl`) works, we trace out one application of it:

```
ssl((5, 3, (2, (1, 4, (6))), 9))
5 + ssl((3, (2, (1, 4, (6))), 9))
5 + 3 + ssl(((2, (1, 4, (6))), 9))
5 + 3 + ssl((2, (1, 4, (6)))) + ssl((9))
5 + 3 + 2 + ssl(((1, 4, (6)))) + ssl((9))
```

Tracing `sumSuperList`

In order to understand how `sumSuperList` (which we shorten to `ssl`) works, we trace out one application of it:

```
ssl((5, 3, (2, (1, 4, (6))), 9))
5 + ssl((3, (2, (1, 4, (6))), 9))
5 + 3 + ssl(((2, (1, 4, (6))), 9))
5 + 3 + ssl((2, (1, 4, (6)))) + ssl((9))
5 + 3 + 2 + ssl(((1, 4, (6)))) + ssl((9))
5 + 3 + 2 + ssl((1, 4, (6))) + ssl(()) + ssl((9))
```

Tracing `sumSuperList`

In order to understand how `sumSuperList` (which we shorten to `ssl`) works, we trace out one application of it:

```
ssl((5, 3, (2, (1, 4, (6))), 9))
5 + ssl((3, (2, (1, 4, (6))), 9))
5 + 3 + ssl(((2, (1, 4, (6))), 9))
5 + 3 + ssl((2, (1, 4, (6)))) + ssl((9))
5 + 3 + 2 + ssl(((1, 4, (6)))) + ssl((9))
5 + 3 + 2 + ssl((1, 4, (6))) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + ssl((4, (6))) + ssl(()) + ssl((9))
```

Tracing `sumSuperList`

In order to understand how `sumSuperList` (which we shorten to `ssl`) works, we trace out one application of it:

```
ssl((5, 3, (2, (1, 4, (6))), 9))
5 + ssl((3, (2, (1, 4, (6))), 9))
5 + 3 + ssl(((2, (1, 4, (6))), 9))
5 + 3 + ssl((2, (1, 4, (6)))) + ssl((9))
5 + 3 + 2 + ssl(((1, 4, (6)))) + ssl((9))
5 + 3 + 2 + ssl((1, 4, (6))) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + ssl((4, (6))) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + 4 + ssl(((6))) + ssl(()) + ssl((9))
```

Tracing `sumSuperList`

In order to understand how `sumSuperList` (which we shorten to `ssl`) works, we trace out one application of it:

```
ssl((5, 3, (2, (1, 4, (6))), 9))
5 + ssl((3, (2, (1, 4, (6))), 9))
5 + 3 + ssl(((2, (1, 4, (6))), 9))
5 + 3 + ssl((2, (1, 4, (6)))) + ssl((9))
5 + 3 + 2 + ssl(((1, 4, (6)))) + ssl((9))
5 + 3 + 2 + ssl((1, 4, (6))) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + ssl((4, (6))) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + 4 + ssl(((6))) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + 4 + ssl((6)) + ssl(()) + ssl(()) + ssl((9))
```

Tracing `sumSuperList`

In order to understand how `sumSuperList` (which we shorten to `ssl`) works, we trace out one application of it:

```
ssl((5, 3, (2, (1, 4, (6))), 9))
5 + ssl((3, (2, (1, 4, (6))), 9))
5 + 3 + ssl(((2, (1, 4, (6))), 9))
5 + 3 + ssl((2, (1, 4, (6)))) + ssl((9))
5 + 3 + 2 + ssl(((1, 4, (6)))) + ssl((9))
5 + 3 + 2 + ssl((1, 4, (6))) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + ssl((4, (6))) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + 4 + ssl(((6))) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + 4 + ssl((6)) + ssl(()) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + 4 + 6 + ssl(()) + ssl(()) + ssl(()) + ssl((9))
```

Tracing `sumSuperList`

In order to understand how `sumSuperList` (which we shorten to `ssl`) works, we trace out one application of it:

```
ssl((5, 3, (2, (1, 4, (6))), 9))
5 + ssl((3, (2, (1, 4, (6))), 9))
5 + 3 + ssl(((2, (1, 4, (6))), 9))
5 + 3 + ssl((2, (1, 4, (6)))) + ssl((9))
5 + 3 + 2 + ssl(((1, 4, (6)))) + ssl((9))
5 + 3 + 2 + ssl((1, 4, (6))) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + ssl((4, (6))) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + 4 + ssl(((6))) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + 4 + ssl((6)) + ssl(()) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + 4 + 6 + ssl(()) + ssl(()) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + 4 + 6 + 0 + 0 + 0 + ssl((9))
```

Tracing `sumSuperList`

In order to understand how `sumSuperList` (which we shorten to `ssl`) works, we trace out one application of it:

```
ssl((5, 3, (2, (1, 4, (6))), 9))
5 + ssl((3, (2, (1, 4, (6))), 9))
5 + 3 + ssl(((2, (1, 4, (6))), 9))
5 + 3 + ssl((2, (1, 4, (6)))) + ssl((9))
5 + 3 + 2 + ssl(((1, 4, (6)))) + ssl((9))
5 + 3 + 2 + ssl((1, 4, (6))) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + ssl((4, (6))) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + 4 + ssl(((6))) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + 4 + ssl((6)) + ssl(()) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + 4 + 6 + ssl(()) + ssl(()) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + 4 + 6 + 0 + 0 + 0 + ssl((9))
5 + 3 + 2 + 1 + 4 + 6 + 0 + 0 + 0 + 9 + ssl(())
```

Tracing `sumSuperList`

In order to understand how `sumSuperList` (which we shorten to `ssl`) works, we trace out one application of it:

```
ssl((5, 3, (2, (1, 4, (6))), 9))
5 + ssl((3, (2, (1, 4, (6))), 9))
5 + 3 + ssl(((2, (1, 4, (6))), 9))
5 + 3 + ssl((2, (1, 4, (6)))) + ssl((9))
5 + 3 + 2 + ssl(((1, 4, (6)))) + ssl((9))
5 + 3 + 2 + ssl((1, 4, (6))) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + ssl((4, (6))) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + 4 + ssl(((6))) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + 4 + ssl((6)) + ssl(()) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + 4 + 6 + ssl(()) + ssl(()) + ssl(()) + ssl((9))
5 + 3 + 2 + 1 + 4 + 6 + 0 + 0 + 0 + ssl((9))
5 + 3 + 2 + 1 + 4 + 6 + 0 + 0 + 0 + 9 + ssl(())
5 + 3 + 2 + 1 + 4 + 6 + 0 + 0 + 0 + 9 + 0
```

Flattening a `SuperLList`

Practice Question: Write the function `flatten` that takes a `SuperLList` of any and returns the *flattened* version of that `LList`. A flattened version of a nested `LList` is a one-dimensional `LList` that contains all the items contained within the original `LList`. The order of items is relative to their order in the original `LList`.

```
flatten(LL(5, 3, LL(2, LL(1, 4), LL(6)), LL(LL(LL(LL(1))))))
  -> LL(5, 3, 2, 1, 4, 6, 1)

flatten(LL(LL(1, 2), LL(3, 4), LL(5, 6)))
  -> LL(1, 2, 3, 4, 5, 6)
```