Train Shunting

Emil Göransson

Spring Term 2023

Introduction

This report will cover the solution to the exercise Train Shunting. It consists of three files: moves.ex, shunt.ex and train.ex.

Train.ex

train.ex a file that consists a bunch of functions that can be used on our desired train. These are: take/2, drop/2, append/2, member/2, position/2, split/2 and main/2.

take/2 looks like the following and is very straight forward:

```
sdef take(_, 0) do
  []
end

def take(trains, n) when n < 0 do
  drop(trains, list_length(trains) + n)
end

def take([h | t], n) do
  [h | take(t, n - 1)]
end
```

The reason I have a special condition for when n¡0 is because my implementation of a function single/2 found inside moves.ex uses it. My first implementation used Enum.find (which has support for n¡0 which i utilized) thus to save time and not change single/2 I implemented a similar function in take/2 that uses drop/2. (I noticed a connection between drop/2 and take/2)

$$take(list, -1) = drop(list, 2) = drop(list, length(list) - 1) \rightarrow$$

$$\rightarrow take(list, -n) = drop(list, length(list) + n)$$

which also applies the other way around.

drop/2 is very similar to take/2 and looks like the following:

```
def drop([], 0) do
  []
end
def drop(list, n) when n < 0 do
  take(list, list_length(list) + n)
end
def drop([h | t], 0) do
  [h | t]
end
def drop([_ | t], n) do
  drop(t, n - 1)
end
```

Here i utilize the same connection as before to define drop for negative n. list_length/1 is a simple list-length counter implemented by me.

append/2 is implemented using the build in append "operator" ++.

member/2 tells us if a object exists within a list or not. My implementation is the following:

```
def member([], _) do
  false
end
def member([h | t], y) do
  if(h == y) do
    true
  else
    member(t, y)
  end
end
```

position/2 returns the first position of a object within a list. the basecase is that if empty list return 0. if there is a list we check if the head is equal to the object we are searching for, if it is we return 1, othersie we use recursion and call position again with the tail and the object and we add one.

split/2 splits the train into two parts, ignoring the object we use as a splitter. it uses both take, position and drop and looks like the following

```
def split(train, y) do
  {take(train, position(train, y) - 1), drop(train, position(train, y))}
end
```

main/2 was the most difficult one to implement but I managed by first pattern matching what we want to return k, remain, take with main(t,n) through recursion. Now we check if k == 0, if it does we stop and return {k, [h — remain], take}, otherwise (if k != 0) we return k-1, remain, [h—take]. What happens here is that the main(t,n) is called until either k == 0 (n) or it reaches its base case where the list is empty. if that is the case we return {n, [], []}

moves.ex

The function single/2 takes a state and a move, and returns a new state based on the move. It takes two arguments, a move in the format :one / :two, n where n is the amount of wagons moved to either track :one or :two. the second argument is the state and has the format main, one, two since the problem consists of three tracks, one main, nr one, nr two.

my implementation of single/2 uses 5 single functions, the fifth and last one is basically a catch-all if the four first ones doesn't match. If we get a move that is not of the from :one, n or :two, n we just ignore it and return the state as it is.

The other four cases uses pattern matching based on two factors based on the format of the move. If the move starts with :one we select either single nr1 or single nr2 depending on if n is positive or negative.

```
    # uses fact that -n arg in .drop reverses order
def single({:one, n}, {main, one, two}) when n >= 0 do
  {Train.drop(main, -n), Train.append(Train.take(main, -n), one), two}
end

# uses fact that -n arg in .drop reverses order
def single({:one, n}, {main, one, two}) when n < 0 do
  {Train.append(main, Train.take(one, -n)), Train.drop(one, -n), two}
end
```

Here we utilize the fact that giving drop or take a negative number (n¡0) reverses the order of how the drop/take works. It allows us to in a simple and straight forward way take, drop and move the last wagon in the train. If we e.g drop the last wagon we want to append the wagon we dropped so it doesn't get lost. Basically Drop and take when given negative numbers allows for this to be easily achieved. I acknowledge that by using main (that I implemented in the train.ex section) the code would become a lot more readable and properly have a more straight-forward implementation since main/2 returns the split wagons in a very convenient form.

sequence/2

The function sequence/2 makes it more convenient to read and apply a larger sum of moves at a time. It takes two arguments, a list of moves and a state. It will then go though every move in the list and return the new state.

Using single/2 we can easily implement the function like the following:

```
def sequence([], state) do
  [state]
end

def sequence([h | t], state) do
  [state | sequence(t, single(h, state))]
end
```

Shunt.ex

Here we implement three functions, find/2, few/2 and rules/1.

find/2

The find/2 function takes two states, a start-state and a end-state. what find/2 will return is a move-set that can be used to go from the start-state to the end-state. Using the numerated steps in the exercises-description the following code could be created.

```
def few(_, []) do
  []
end
def few([h | t], [y | ys]) do
  ## checks if we already have wagon1 in right position
  if(h == y) do
    few(t, ys)
  else
    {hs, ts} = Train.split([h | t], y)
    moves = [
      {:one, Train.list_length(ts) + 1},
      {:two, Train.list_length(hs)},
      {:one, -(Train.list_length(ts) + 1)},
      {:two, -Train.list_length(hs)}
    ]
    Train.append(moves, few(Train.append(ts, hs), ys))
  end
end
```

few/2

The function few/2 is a optimized find/2 function that checks if the head of the list already is in the right position. It works though a If-statement in the beginning of the second few-function that checks if the head of the initial-state is the same as the head of the end-state. If that is the case we don't need to move the head anymore and because of this we instead rerun the code by calling few again with only tail of the initial state, ignoring the head. If the head of the initial-state is never the same as the head of the end-state few/2 will behave exactly the same as find/2, only being a bit slower because of the if statement in the beginning.

rules/1

The last part of the train-problem is to implement a rules/1 function that further simplifies the find/2 function. We do this by utilizing pattern matching. The code is the following:

```
def rules([]) do
  []
end
def rules([{:one, n} | [{:one, m} | t2]]) do
  rules([{:one, n + m} | rules(t2)])
end
def rules([{:two, n} | [{:two, m} | t2]]) do
  rules([{:two, n + m} | rules(t2)])
end
def rules([{:one, 0} | t]) do
  rules(t)
end
def rules([{:two, 0} | t]) do
  rules(t)
end
def rules([h | []]) do
  [h]
end
def rules(list) do
  list
end
```