Lab 11 - Train shunting

Weichen Chai

Spring 2023

Introduction

This paper seeks to answer the questions presented in"Train shunting" for ID1019. In this assignment, we will explore how the functional language Elixir can be utilized to solve the train shunting puzzle problem, working with processing of lists, and recursion.

Tasks 1

The train shunting problem asks for us to arrange the sequence of wagons into a given order with "shunting stations", where wagons will be presented in atom form, and description of start for shunting station will be done by three different lists. moves are performed in the form of binary tuple, where the elements are :one or :two, representing the list describing tracks. Followed by this element, there will be an integer, which indicates the number of wagons that moves from or to a shunting station, where if the integer is positive, it is moving to the shunting station and if it is negative then the opposite happens.

0.1 Trains

We start by first writing out the train processing routines, which can be done by first making a train module. This module will include functions such as take, drop append etc. which are all cases which the train could perform. The code looks something like the following.

```
def take(_, 0) do [] end
def take([], _) do [] end
def take([head | tail], n) when n > 0 do [head | take(tail, n - 1)] end
//returns n wagons of the train.
...
def member([], _) do false end
def member([y | _], y) do true end
```

```
  def member([_| tail], y) do member(tail, y) end
  // memeber checks if an element is a member of the train
...
// Most importantly, there is main, where wagons are in reverse order
//leftmost position is for first wagon
Def main([head | tail], n) do
    case main(tail, n) do
      {0, drop, take} ->
        {0, [head | drop], take}
      {n, drop, take} ->
        {n - 1, drop, [head | take]}
    end
  end
```

0.2 Moves

After completing the train module, we begin with moves, which describes the moves that can be performed on the train wagons. In this assignment, we start by implementing a "single" function that returns a computed state of inputted moves.

```
// no wagons moved
def single({_, 0}, state) do state end
def single({:one, n}, {main, one, two}) when n > 0 do
// from main to track 1 (right)
    {0, remain, wagons} = Train.main(main, n)
    {remain, Train.append(wagons, one), two}
  end
...//from track 1 to main (left)

 def single({:two, n}, {main, one, two}) when n < 0 do
 // track 2 to main (left)
    wagons = Train.take(two, -n)
    {Train.append(main, wagons), one, Train.drop(two, -n)}
  end
...// Main to track 2 (right)
```

The second task given under this section asks for us to implement a "sequence" function that takes states as inputs as well as moves, so that it gives us an indication of how moves are made after running this module.

```
...//base case
def sequence([head | tail], state) do
// state contains current status of train
```

```
    [state | sequence( tail, single(head, state ))] end
```

Thus this section is complete with the contents stored inside moves.ex

0.3 Shunting problem

In this section, we are asked to find sequence of moves that change the wagons' order on the main track. We first finish a find function which takes in trains xy and ys. Following the descriptions given in the assignment, we can start by writing the following:

```
...//base case,no moves
  def find(xs, [y | ys]) do
    {hs, ts} = Train.split(xs, y) [
    // hs are wagons before y ts are wagons after y
      {:one, length(ts) + 1}, //main to track1 (right)
      {:two, length(hs)}, //hs to track 2(left)
      {:one, -(length(ts) + 1)}, //track 1 to main but reverse order
      {:two, -length(hs)} | find(Train.append(hs, ts), ys)
      //track 2 to main in order.
    ]
  end
```

Testing with the example given, after compiling all 3 modules, we perform the find function the result is:

```
shuntT.find([:a,:b],[:b,:a])
// result:
[one: 1, two: 1, one: -1, two: -1, one: 1, two: 0, one: -1, two: 0]
```

Which is exactly the same as the answer shown in the assignment.

However, since there are a lot of redundant moves, we have to somehow simplify it with a new function, which is an improved version of din, it will look similar, with essentially the only changes being to change the xs in the function above to hs.

0.4 Move Compression

Looking for further optimization, we use the compress function given in the assignment, and alongside it, write rules procedures which applies rules recursively, Compress function looks like the following:

```
 def compress(ms) do
    ns = rules(ms)
    if ns == ms do
      ms
```

```
    else
      compress(ns)
    end
  end
```

The rules that will be included here are as following: 1. Replace :one,n directly followed by :one,m with :one,n+m. 2. Replace :two,n directly followed by :two,m with :two,n+m.

And in code recursive form, they should be:

```
...// base case
  def rules([{_, 0} | tail]) do rules(tail) end
  def rules([{:one, n}, {:one, m} | tail]) do rules([{:one, n + m} | tail]) end
  def rules([{:two, n}, {:two, m} | tail]) do rules([{:two, n + m} | tail]) end
  def rules([head | tail]) do [head | rules(tail)] end
```