Train shunting

Sareh Jalalizad

Spring Term 2023

Introduction

This programming task involves solving the train shunting problem by moving wagons between three tracks: the main track, and two other tracks referred to as “one” and “two”. The task is to implement a program that determines the optimal sequence of moves that will result in the desired configuration of wagons on the tracks.

Some train processing

The first part of the task was to create a module called ”Train” which is responsible for processing and manipulating lists. The module includes seven different recursive functions: take, drop, append, member, position, split, and main.

Take function takes the first n wagons of the train and returns them. It does this recursively by taking the head of the list and adding it to the result of recursively calling take on the tail of the list, until n wagons have been taken or the list is empty. The base case for this recursion is when n is 0, in which case an empty list is returned. The drop function drops the first n wagons of the train and returns the remaining wagons. The third function is append which returns the train that is the combination of the two trains. *member/2* checks whether a wagon is a member of a train or not, and *position/2* returns the position of a wagon in a train. The last one is *main/2* which takes a train and a number n and returns a tuple {k, remain, take} where k is the number of wagons remaining to have n wagons in the taken part, remain is a list of the wagons that were not moved, and take is a list of the wagons that were moved.

```
def take(_, 0) do [] end
def take([], _) do [] end
def take([h|t], n) when n > 0 do [h | take(t, n-1)] end

def drop(train, 0) do train end
```

```
def drop([], _) do [] end
def drop([_|t], n) when n > 0 do drop(t, n-1) end

def append([], train) do train end
def append(train, []) do train end
def append([h1|t1], train2) do [h1 | append(t1, train2)] end

def main(train, 0) do {0, train, []} end
def main(train, n) do
  case train do
    [] -> {n, [], []}
    [h|t] ->
      case main(t, n) do
        {0, remain, take} ->
            {0, [h|remain], take}
        {k, remain, take} ->
            {k-1, remain, [h|take]}
      end
   end
end
```

In this implementation, we first handle the base cases where either the train is empty or n is 0. If the train is empty, we just return the remaining number of wagons, and two empty lists for the remaining wagons and taken wagons. If n = 0, we return 0 and the original train for the remaining wagons, and an empty list for the taken wagons.

If neither of the base cases applies, we recursively call the main/2 function with the tail of the train and the remaining number of wagons to be taken. We then pattern match on the result of the recursive call. If we have already taken the required number of wagons, we add the current wagon to the remaining wagons and return the same taken wagons as before. If we have not yet taken the required number of wagons, we add the current wagon to the taken wagons and decrement the number of wagons remaining to be taken. We return the new number of wagons remaining to be taken, the remaining wagons, and the new taken wagons.

Applying moves

The next part of the task is to create a module called "Moves" that includes two functions: *single/2* and *sequence/2*. The *single* function receives a move and an input state, applies the move, and then returns the new state. On the other hand, *sequence* takes a list of moves and a state as input and returns a list of states representing the transitions when the moves are performed.

```
def single({_, 0}, {main, one, two}) do {main, one, two} end
 def single({track, n}, {main, one, two}) do
   case track do
     :one ->
       cond do
         n > 0 ->
           {0, remain, take} = Train.main(main, n)
           {remain, Train.append(take, one), two}
         n < 0 ->
           take = Train.take(one, -n)
           {Train.append(main, take), Train.drop(one, -n), two}
       end
     :two ->
       cond do
         n > 0 ->
           {0, remain, take} = Train.main(main, n)
           {remain, one, Train.append(take, two)}
         n < 0 ->
           take = Train.take(two, -n)
           {Train.append(main, take), one, Train.drop(two, -n)}
       end
   end
 end

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

The *single* function takes a tuple {track, n} that specifies a move to be made on a given track (:one or :two) and an input state with three tracks (main, one, two). To implement this, the function pattern-matches on the track parameter to decide which track is involved and what the different elements of the state are. It then uses Train.main/2 to move the wagons on the given track according to the value of n.

The shunting problem

The next part of the task was to implement Shunt module which provides different functions: find, few, and compress. The function find takes two trains xs and ys as input, and returns a list of moves that transform the state {xs, [], []} into {ys, [], []}. In the general case, the function takes the leftmost element y from ys, splits the input train xs into two parts hs

and ts such that hs contains all the wagons before y and ts contain all the wagons after y, and computes a sequence of moves that moves the wagon y to the leftmost position on the main track. These moves are stored in the variable moves and involve moving the wagons on tracks one and two to the main track. Finally, the function recursively computes the sequence of moves required to transform the updated input train Train.append(hs, ts) into the remaining output train ys, and appends this sequence of moves to the list moves. The final list of moves required to transform xs into ys is obtained by concatenating all the lists of moves computed in each recursive call.

The few/2 function works in a similar way to the find/2 function, but it considers whether the next wagon is already in the correct position for each recursive call. If the wagon is already in the right position, no moves are needed.

```
def find([], []) do [] end
def find(xs, [y | ys]) do
  {hs, ts} = Train.split(xs, y)
  [
    {:one, length(ts) + 1},
    {:two, length(hs)},
    {:one, -(length(ts) + 1)},
    {:two, -length(hs)} | find(Train.append(hs, ts), ys)
   ]
 end

def few([], []) do [] end
def few([y | xs], [y | ys]) do few(xs, ys) end
def few(xs, [y | ys]) do
  {hs, ts} = Train.split(xs, y)
  [
    {:one, length(ts) + 1},
    {:two, length(hs)},
    {:one, -(length(ts) + 1)},
    {:two, -length(hs)} | few(Train.append(hs, ts), ys)
  ]
end
```