ID1019: Train shunting

Alexander Lundqvist

Spring Term 2023

Introduction

In this assignment we will attempt to solve to a train shunting problem. The problem involves re-arranging a train of wagons into a desired configuration using a shunting station that includes a "main" track and two shunting tracks. The wagons are represented by atoms (`:a`,`:b`,`:c`, ...) and the trains are represented as lists of wagons (`[:a, :b, :c, ...]`). We also have the current state of the tracks at the station which is represented as {`:main, :track_1, :track_2`}.

To achieve the desired train configuration we need to implement a solution that finds a short sequence of moves that changes the configuration of the wagons on the `:main` track. A move is represented by a binary tuple where the first element is the track and the second element is an integer that represents the number of wagons, for example {`:track_1, 3`}.

Method

The assignment specifies that the solution should have three different modules named `Train`, `Moves` and `Shunt`. The `Train` module represents an instance of a train and contains functions for manipulating its list of wagons. As we are not allowed to use any existing modules in Elixir for list processing we will define our own according to the instructions. The implementation of `main/2` will be discussed in the result section.

- `take(train, n)`, which takes the first n wagons of the train and returns them as a new train.
- `drop(train, n)`, which drops the first n wagons of the train and returns the remaining wagons as a new train.
- `append(train_1, train_2)`, which appends the wagons in `train_1` the end of `train_2` and returns the resulting train.
- `member(train, wagon)`, which checks if the `wagon` is a part of the `train` and returns true if it is and false otherwise.

- `position(train, wagon)`, which returns the position of the `wagon` in the the `train`.
- `split(train, wagon)`, which returns a tuple with two trains where the first contains all the wagons before the specified `wagon` and the second contains all the wagons after `wagon`.
- `main(train, wagon)`, which returns the tuple k, remain, take where remain and take are the wagons of train and k are the numbers of wagons remaining to have n wagons in the taken part.

The `Moves` module represents the moves we want to perform on the trains and should define the functions `single/2` and `sequence/2` as specified in the assignment.

- The `single/2` function takes a move and an input state (where the state is a list of three lists representing the train tracks) and return a new state after the move has been performed. The function will use pattern-matching to account for every track and wagon variations.
- The `sequence/2` applies a list of moves to an initial state, and returns a list of states representing the intermediate states after each move is applied.

Finally, we have the `Shunt` module which contains the logic for solving the actual problem. We define a function `find/2`, which takes two trains as input and returns a list of moves that transforms the initial state to the desired state. The instructions also dictates that we try to optimize the `find/2` by removing redundant moves. This will be done in a function `few/2` which will be mentioned in the result section.

The implementation that we've just described is available at this GitHub repository.

Result

In the assignment we we're supposed to implement a recursive function `main/2` without using any other functions in the `Trains` module. This is how I have implemented the function.

```
def main([], n) do {n, [], []} end
def main([head|tail], n) do
  case main(tail, n) do
    {0, remain, take} -> {0, [head|remain], take}
    {n, remain, take} -> {n-1, remain, [head|take]}
end
```

It takes two arguments: a train (i.e. a list of wagons, represented as `[head|tail]`) and a number `n` that represents the number of wagons to move. The function starts by recursively calling itself on the tail of the train until the base case is reached, which is when the end of the train is reached or the number of wagons to move has been reached. The base case returns the tuple {`n, remain, take`} which represents the remaining wagons (`remain`) and the wagons to be moved (`take`), where `n` is the number of wagons that should be left on the main track to have `n` wagons in the `take` part.

The function then checks if `n` is zero, which means that we have found all the wagons that we need to move, so it prepends the current wagon (`head`) to the `remain` part of the tuple and returns {`0, [head|remain], take`}. Otherwise, it decrements `n` by 1 and returns {`n-1, remain, [head|take]`} which means that we still need to move `n-1` wagons, and we should prepend the current wagon (`head`) to the `take` part of the tuple.

0.1 Shunting

As mentioned earlier we have a function `find/2` in the `Shunt` module that allows for rearranging the trains at the station. To improve the function we defined a new function `few/2`, that removes redundant moves. It does this by accounting for wagons that are already in the correct position. The strategy is to move as many wagons as possible in each move and only uses moves that move wagons back and forth between the main track and the parallel shunting track if necessary.

```
def few(train_1, [head|tail]) do
    {wagons_before, wagons_after} = Train.split(train_1, head)
    amount_before = length(wagons_before)
    amount_after = length(wagons_after)

    moves = [
      {:track_1, amount_after + 1},
      {:track_2, amount_before},
      {:track_1, -(amount_after + 1)},
      {:track_2, -amount_before}
    ]

    if amount_before == 0 do
      [] ++ few(wagons_after, tail)
    else
      moves ++ few(wagons_after ++ wagons_before, tail)
    end
  end
```

We also received a `compress/1` function that combines adjacent moves

of the same type into a single move. It is dependent on a function `rules/1`, which simplifies and optimizes a list of moves.`compress/1` calls this function repeatedly until no further simplifications can be made.`rules` was implemented like this:

```
def rules([]) do [] end
def rules([{track, step} | moves]) do
  cond do
    step == 0 ->
      rules(moves)

    moves != [] ->
      [{next_track, next_step} | next_moves] = moves
      cond do
        track == :track_1 && next_track == :track_1 ->
          [{:track_1, step + next_step}] ++ rules(next_moves)
        track == :track_2 && next_track == :track_2 ->
          [{:track_2, step + next_step}] ++ rules(next_moves)
        true ->
          [{track, step}] ++ rules(moves)
      end
    true ->
      [{track, step}] ++ rules(moves)
  end
end
```

The rules work like this:

- If a move has a step size of 0, then it can be removed.
- If two consecutive moves operate on the same track, they can be combined into a single move with a step size equal to the sum of the step sizes of the original moves.

The `rules/1` function applies these rules to the input list of moves by looking at each move in turn and either removing it (if it has a step size of 0) or combining it with the next move (if they operate on the same track). The resulting list of moves is then returned.