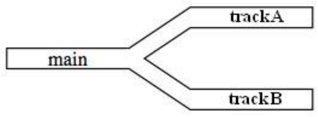
TDT4165 PROGRAMMING LANGUAGES

Fall 2015

Exercise 02
Train-shunting

Problem description

You're in charge of shunting a train. We assume that each train car has an engine, and that there are no locomotives. The input is the current sequence of cars, and the desired sequence. The problem is to rearrange the cars, using moves in a station, in such a way that you end up with the desired sequence. The station is shown in figure 1, and a situation in the station is called a state. A move describes how the cars move from track to track.



Figur 1: Station

Goal

The goal of this problem is to find a short sequence of moves that changes the initial sequence of cars on the main track to the desired sequence on the main track. Before attempting this, we need to model the problem, and develop some help for dealing with lists.

The exercise objective

In this exercise we look at some important things: How do we model a problem using datastructures like lists and records? How do we work with lists? This ranges from simple to more complicated patterns of recursion. It is in addition a good introduction to Oz and Mozart. We have also made a program to visualize your algorithm, hopefully making it a little easier to understand how it works.

Modeling

Trains, cars and states

Cars are modeled as atoms, and a train on the track as a list of atoms. A train has no duplicate cars (meaning [a b] is a valid train, but [a a] is not). A complete description of the state includes three lists; one for each of the tracks ("main", "trackA" and "trackB"). A complete state is the a record with `main`, `trackA` and `trackB` as features with corresponding lists as fields. In addition, the label of the record should be `state`.

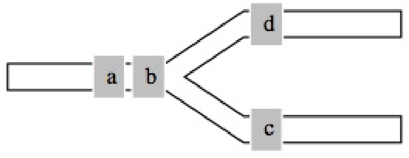


Figure 2: Example of a state

The state `state(main:[a b] trackA:[d] trackB[c])` is shown in Figure 2.

Moves

A move has a value stored as a field in a tuple with the label `trackA` or `trackB`. (Tuples as the same as record, but without specified features.) Each move includes a number, which indicates how many cars are being moved. Ex: `trackA(2), trackB(2)` and `trackA(~3)` are all moves.

Executing a move given a state

Moves describe how a state is changed.

- If the move is `trackA(N)` and N is larger than zero, then the N cars farthest to the right moved from "main" to "trackA".
- If the move is `trackA(N)` and N is lesser than zero, then the N cars farthest to the left moved from "trackA" to "main".
- The moves `trackA(0)` and `trackB(0)` has no effect.

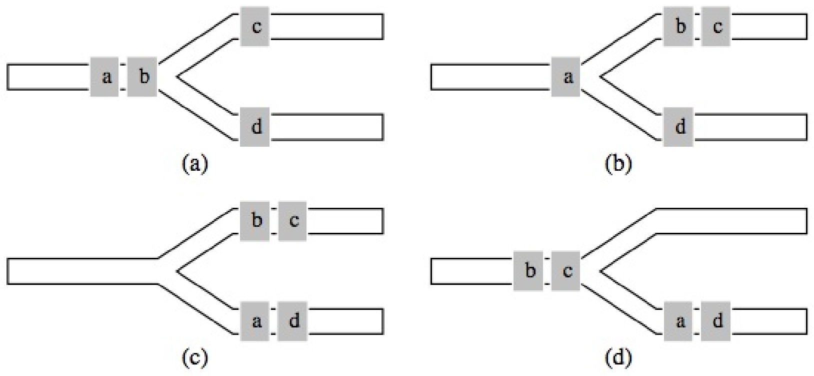


Figure 3: Some moves executed on states

Example

Figure 3 shows som examples of moves executed on states. (a) is the start state, (b) is after `trackA(1)`, (c) after `trackB(1)` and finally (d) after `trackA(~2)`.

Task 1 Lists

First, you shold make som routines for modifying lists.

- {Length Xs} returns the length of Xs
- {Take Xs N} returns a list containing the first N elements of Xs. If N is larger than the size of Xs, return Xs.
- {Drop Xs N} returns Xs without the N first elements. If N is larger than the size of Xs, return `nil`.
- {Append Xs Ys} returns a list containing all elements of Xs, followed by all elements of Ys.
- {Member Xs Y} returns true if Xs contains Y, false if not.
- {Position Xs Y} returns Y's position in Xs. You can assume Xs contains Y.

If you did Exercise 1, you can use these implementations. Put the implementation in `List.oz`, and include it in your program by writing \insert 'List.oz' before the code that uses them.

Task 2 Apply Moves

The first task is write the `ApplyMoves` function, which takes the initial state and a list of moves as input. The function should return a list with all the different states, with the first being the initial state.

Example:

```
{ApplyMoves state(main:[a b] trackA:nil trackB:nil) [trackA(1) trackB(1) trackA(~1)]}
```

returns the list

```
[state(main:[a b] trackA:nil trackB:nil)
state(main:[a] trackA:[b] trackB:nil)
state(main:nil trackA:[b] trackB:[a])
state(main:[b] trackA:nil trackB:[a])]
```

Visualize states

To get a better understanding of states, you can use the included program `Visualizer.zip`, which can be downloaded from it's learning. To use it, it must be unzipped in your programs directory and included using the line; `\insert 'Visualizer.oz'`. You run it by calling the procedure `Visualize` on a list of states. It can be tested on the example above.

Structure

We can now implement `ApplyMoves`.

- Apply each move recursively.
- If the list of moves is empty (i.e no more moves), return a list containing only the initial state.
- For each move, the program should use pattern recognition to decide which track is involved.
- If S is a state, you can get the list of cars on the main track by calling `S.main`.

The structure of `ApplyMoves` could look something like this:

```
fun {ApplyMoves S Ms}
   %% S is the state, Ms the list of moves to be applied
   %% Returns list of resulting states
   case Ms of nil then ...
   [] M|Mr then
      %% Compute S1 as new state
      S1 = case M of trackA(N) then
      if ... then ...  else ...  end
   [] trackB(N) then
      if ... then ...  else ... end
   end
   in
      ...
   end
end
```

Save your program in a file named `ApplyMoves.oz`, and don't forget to use the visualization program to make sure your program is working correctly.

Task 3 Finding moves

Write a function `Find`, which takes to trains Xs and Ys as input and returns a list of moves transforming the state:

```
state(main:Xs trackA:nil trackB:nil)
```

to

```
state(main:Ys trackA:nil trackB:nil)
```

You can assume that Xs and Ys are permutations of eachother, and that each train car is unique.

- The problem should be solved recursively, and each step should move a car to a position demanded by Ys.
- Base case is simple. If there are no cars, no move is needed.
- If not, take the first car Y from Ys and move it to the front of the main track. This is done by finding a sequence of moves:
  - Split Xs into Hs (head) and Ts (tale) where Hs is the cars before Y in Xs and Ts is the cars after Y in Xs. Write a function `SplitTrain` which takes the list Xs and the car Y and returns the pair Hs#Ts. Example:

    `{SplitTrain [a b c] a}= nil#[b c]`

    Use the functions `Position, Take` and `Drop`.
  - Move Y and the following cars (Ts) to "trackA".
  - Move the rest of the cars (Hs) to "trackB".

  - Move all the cars from ”trackA” to ”main”.
  - Move all the cars from ”trackB” to ”main”.

- After moving a car to it’s correct position, we only need to look at the rest of the cars of both Xs and Ys (using the new sequence on ”main”).

Save your program in a file named `Find.oz`.

Visualizing moves

You should use the visualization program to makes sure your program is correct.

Example

Given input [a b] and desired output [b a], the list of moves returned by `Find`:

```
[trackA(1) trackB(1) trackA(~1) trackB(~1)
trackA(1) trackB(0) trackA(~1) trackB(0)]
```

Task 4 Find less moves

Make the function `FewFind` which behaves like `Find`, but for each recursive operation checks if the next car is already in the correct position. If it is, no moves are required. Only a few modifications to the program `Find` are needed. You should save the program in a file named `FewFind.oz`.

Example

Given input [c a b] and desired output [c b a], the list of moves returned by `FewFind` should be:

```
[trackA(1) trackB(1) trackA(~1) trackB(~1)]
```

Task 5 Even fewer moves

We are still not happy with `FewFind`, and want to optimize it by adding the following rules:

- Replace `trackA(N)` directly followed by `trackA(M)` with `trackA(N+M)`.
- Replace `trackB(N)` directly followed by `trackB(M)` with `trackB(N+M)`.
- Remove `trackA(0)`.

- Remove `trackB(0)`.

Make a function `Compress` which takes a list of moves as input and returns a compressed list of moves.

- Make a function`ApplyRules` which runs through the rules recursively.
- The run `ApplyRules` again until the list of moves stops changing.

`Compress` wil look something like:

```
fun {Compress Ms}
   Ns={ApplyRules Ms}
in
   if Ns==Ms then Ms else {Compress Ns} end
end
```

Save the program in a file named `Compress.oz`.

Task 6 Find really few moves

The problem with both `Find` and `FewFind` is that you always move all the cars from "trackA" and "trackB" back to "main" even if there exist opportunities to use "trackA" and "trackB" to move the cars into the correct position. We want to exploit this. Make a function `FewerFind`, which takes four arguments: Ms for the cars on "main", Os for the cars on "trackA", Ts for the cars on "trackB" and Ys for the desired train. `FewerFind` is a recursive function, and each recursive call moves car Y from Ys to its correct position. What is new, is that a car can be on any of the tracks.

- If Ms contains Y: Move Y as before, but leave the other cars on "trackA" and "trackB"
- If Os contains Y: Move the cars in front of Y to "main", and then to "trackB". Then move Y to its desired position.
- Do the same if Ts contains Y.

When `FewerFind` is called for the first time, Os and Ts are empty lists. Example:

```
{FewerFind [a b] nil nil [b a]}
```

returns

```
[trackA(1) trackB(1) trackA(~1) trackB(0) trackA(0) trackB(~1)]
```

Save your program in a file named `FewerFind.oz`.