Assignment 1, CS2104: Train Shunting

Due date: 2003-09-05
Points: 15 out of 100 points

Problem Description

You are in charge of shunting wagons of a train. In the following we assume that each wagon is self driving and that the train has no explicit engine.

The description for your shunting task is given by two sequences of wagons: the given train and the desired train. Your task is to rearrange the given train with help of your shunting station such that the desired train is obtained. You are not only supposed to rearrange the train but also to compute a sequence of shunting moves (which are called just "moves" from now on).

The shunting station is shown in Figure 1. It has a "main" track and two shunting tracks "one" and "two". A situation in the shunting station is called *state*. A *move* describes how wagons move from one track to another.

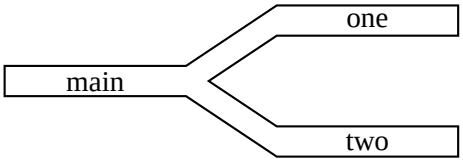

Figure 1: Train shunting station.

Goal. Our ultimate goal in this assignment is to find a short sequence of moves that turn a train on the track "main" into another configuration of the train on "main".

Before we attempt this goal we will fix the modeling of our problem and develop some list processing support.

Assignment Purpose. This assignment exercises several important issues. How are problems modelled by data structures such as lists and records. How are lists processed; This ranges from simple to more complicated patterns of recursion over lists. This assignment is of course also geared at getting you started with Oz and Mozart.

And last, but not least, we hope that you have *some fun* solving this little puzzle. As a potential fun increaser, we wrote a tiny train station visualizer for you that will allow you to graphically check what your programs actually do!

Points. This assignment will give you 15 percent of the course requirements. You will get 15 points if you submit at latest at the due date!

Modelling

Trains, Wagons, and States. Wagons are modelled as atoms and trains on tracks as lists of atoms. A train has no duplicate wagons (that is, `[a b]` is a train, whereas `[a a]` is not).

A complete description of the state of a shunting station consists of three lists: a list describing the train on track "main", and two lists describing tracks "one" and "two". An entire state is represented as a record that has features `main`, `one`, and `two` with the corresponding lists as fields for the trains on the tracks. Additionally, we require that the label of a state is `state`.

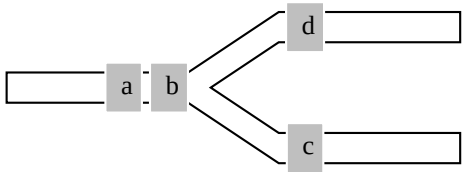

Figure 2: Example state displayed.

The state

```
state(main:[a b] one:[d] two:[c])
```

is visualized in Figure 2.

Moves. A move is a unary tuple (that is, a record that has the single feature `1`). The label of a move is either `one` or `two`. The single field of a move is an integer. For example, `one(2)`, `two(2)`, and `one(~3)` are all moves.

Applying a Move to a State. Moves describe how one state is transformed into another:

- If the move is `one(N)` and `N` is greater than zero, then the `N` right-most wagons are moved from track "main" to track "one".

  If there are more than `N` wagons on track "main", the other wagons remain.

- If the move is `one(N)` and `N` is less than zero, then the `N` left-most wagons are moved from track "one" to track "main".

  If there are more than `N` wagons on track "one", the other wagons remain.

- The move `one(0)` has no effect.

The same holds true for moves with label `two` concerning track "two".

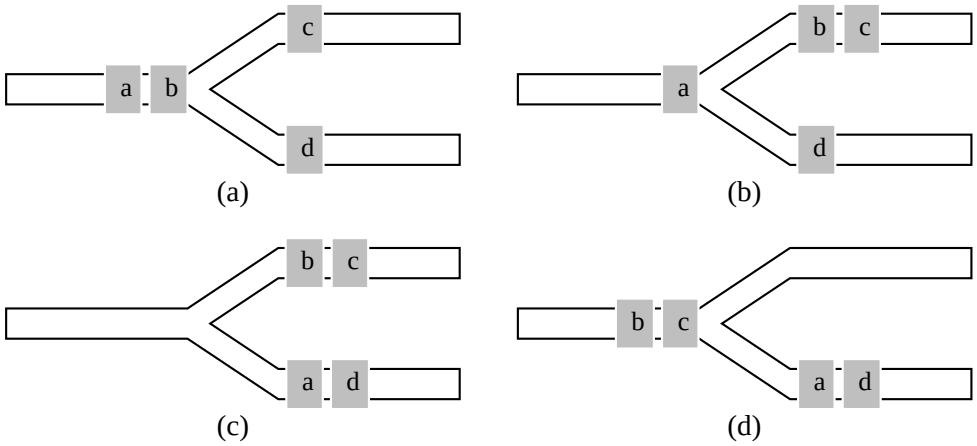

Figure 3: Moves applied to states.

Example. Figure 3 shows examples of moves applied to states, where (a) is the initial state, (b) after application of `one(1)`, (c) after application of `two(1)`, and finally (d) after application of `one(~2)`.

1 Some List Processing

Before we actually start, we will develop some list-processing routines that you will need later[1].

1. `{Length Xs}` returns the length of the list `Xs`.
2. `{Take Xs N}` returns the list containing the first `N` elements of `Xs`. If the length of `Xs` is less than `N` it returns `Xs`.
3. `{Drop Xs N}` returns the list `Xs` without its first `N` elements. If the length of `Xs` is less than `N` it returns `nil`.
4. `{Append Xs Ys}` returns the list `Xs` with the elements of `Ys` appended.
5. `{Member Xs Y}` tests whether `Y` is an element of `Xs`.
6. `{Position Xs Y}` returns the first position of `Y` in the list `Xs`. You can assume that `Y` is an element of `Xs`.

   For example, `{Position [a b c] b}` returns `2`.

Please put all together in one file `List.oz`. You can include the program in that file by putting the line

```
\insert ´List.oz´
```

before the procedures that use these list-processing procedures.

[1] We know that they are predefi ned in Mozart, but just see it as warm up.

2 Applying Moves

The first task is writing a procedure `ApplyMoves` that takes an initial input state and a list of moves. It results in a list of states. If the list of moves has n elements, the resulting list of states has $n + 1$ elements. Its first element is the initial state and its last element is the final resulting state.

For example,

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

returns the list of states

```
[state(main:[a b] one:nil two:nil)
 state(main:[a]   one:[b] two:nil)
 state(main:nil   one:[b] two:[a])
 state(main:[b]   one:nil two:[a])]
```

Visualizing States. To get a better understanding of states, you can use the Visualizer. In order to use the visualizer you have to download the program `Visualizer.oz` from the course homepage.

To use the Visualizer in your program, copy the file into your current working directory, and feed in Mozart the line

```
\insert ´Visualizer.oz´
```

If you like, you can download another version of the visualizer `Visualizer.zip` that offers the same functionality as the first one with a slightly nicer interface. In this case you got several files (`Visualizer.oz` , and a few image-files). Unzip the file (`Visualizer.zip`) into your current working directory. Make sure, if not already the case, that the image files are stored in a subdirectory called (`images`) after that the usage is the same for both versions.

After you have done that, you can apply the procedure `Visualize` to a *list* of states. So try visualizing the above example by:

```
{Visualize [state(main:[a b] one:nil two:nil)
            state(main:[a]   one:[b] two:nil)
            state(main:nil   one:[b] two:[a])
            state(main:[b]   one:nil two:[a])]}
```

The “Prev” and “Next” buttons allow you to step through the states.

Structure. Now we are in a position to develop `ApplyMoves`. Approach the task as follows:

- Recursively apply each move in turn.
- When no moves are to be applied, our specification says that the initial state is to be returned (base-case).
- For each move, your program should decide by pattern-matching which track is involved.

- For each track, you have to decide whether wagons are moved *to* or *from* the track (that is, is `N` positive or negative).
- Taking wagons from the end of a train can be done by using `Length` and `Drop`; taking wagons from the beginning of a train can be done by using `Take`; adding wagons to a track can be done by `Append`.
- If `S` is a state, the wagons on track "main" are accessed by `S.main`, and so on.

The structure of `ApplyMoves` thus looks as follows:

```
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 one(N) then
              if ··· then ··· else ··· end
           [] two(N) then
              if ··· then ··· else ··· end
           end
   in
      ···
   end
end
```

Please store your program in a file called `ApplyMoves.oz`. Do not forget to use the Visualizer to convince yourself that your solution works!

3 Finding Moves

Develop a procedure `Find` that takes two trains `Xs` and `Ys` as input and returns a list of moves, such that the moves transform the state

```
state(main:Xs one:nil two:nil)
```

into

```
state(main:Ys one:nil two:nil)
```

In the following, we require that `Xs` and `Ys` contain the same elements (wagons) and that each wagon is unique (in other words, `Xs` and `Ys` are permutations of each other).

Approach the problem as follows. The problem is solved recursively and each recursive step will move one wagon in the position as required by `Ys`.

The base-case is simple. If there are no wagons, no moves are needed.

Otherwise, we take the first wagon `Y` from `Ys` (the desired train). Our goal is to find a list of moves that takes the wagon `Y` in front position on the main track. This is done by the following moves:

1. Split the train `Xs` into the wagons `Hs` (for head) and `Ts` (for tail), where `Hs` are the wagons before `Y` in `Xs`, and `Ts` are the wagons after `Y` in `Xs`.

   Write a procedure `SplitTrain` that takes a list of wagons `Xs` and the wagon `Y` and returns the pair `Hs#Ts`.

   For example,

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

   Use the procedures `Position`, `Take`, and `Drop`.

2. Move `Y` and the following wagons (that is `Ts`) on track "one".

3. Move the remaining wagons (that is, `Hs`) on track "two".

4. Move all wagons on "one" to "main" (this includes `Y`, which goes as needed to the front of "main").

5. Move all wagons on "two" to "main".

After having moved one wagon in the right position, we need only consider the remaining wagons of both `Xs` and `Ys` (of course in the new order as they are now on the main track!).

Please store your program in `Find.oz`.

Visualizing Moves. You can of course use the Visualizer to gain confidence! Use `ApplyMoves` to create the list of states for display.

Example. Given the input train `[a b]` and the output train `[b a]`, the list of moves computed by `Find` is:

```
[one(1) two(1) one(~1) two(~1)
 one(1) two(0) one(~1) two(0)]
```

4 Finding Less Moves

Develop a procedure `FewFind` that behaves as `Find` but that takes for each recursive application into account whether the next wagon is already in the right position. If so, no moves are needed.

Proceed by modifying (only very few modifications are needed) your program for `Find`.

Please store your program in `FewFind.oz`.

Example. Given the input train `[c a b]` and the output train `[c b a]`, the list of moves computed by `FewFind` is:

```
[one(1) two(1) one(~1) two(~1)]
```

5 Move Compression

The list of moves computed by `FewFind` is still awkward and can be easily optimized according to the following rules:

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)`.
3. Remove `one(0)`.
4. Remove `two(0)`.

These optimizations are *correct* in the sense that the shorter list of moves will compute the same final state.

This task is actually tricky: think for example of

```
[two(~1) one(1) one(~1) two(1)]
```

Repeatedly applying the rules from above actually results in no moves at all. By application of Rule 1 we obtain `[two(~1) one(0) two(1)]`; by Rule 3 `[two(~1) two(1)]`; by Rule 2 `[two(0)]`; and finally by Rule 4 `nil`.

Develop a procedure `Compress` that takes a list of moves and returns a compressed list of moves. Compression must be complete, that is, none of the above rules should be applicable to the returned list of moves.

Approach this task as follows. Develop a procedure `ApplyRules` that applies rules recursively. Then repeat application of `ApplyRules` until the list of moves does not change. Thus, `Compress` is implemented as follows:

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

Please store your program in a file called `Compress.oz`.

6 Finding Really Few Moves

The problem with both `Find` and `FewFind` is that they always push back the wagons from “one” and “two” to “main”, even though there might be some opportunity to actively use track “two” to push wagons from track “one” into position and vice versa. In the following, we are going to take adavantage of this.

Develop a procedure `FewerFind`, that takes four arguments: `Ms` as the wagons on “main”, `Os` as the wagons on “one”, `Ts` as the wagons on “two”, and `Ys` as the desired train.

`FewerFind` works recursively and as before, each recursive invocation will bring the first wagon `Y` of `Ys` into the right position. What is new, is that this wagon might be on either track:

- If `Y` is a member of `Ms`, bring it in the right position as done previously. Leave the other wagons on track “one” and “two”.
- If `Y` is a member of `Os` (it is on track “one”), move the wagons in front first to “main” and then to “two”. Then move `Y` into position. Otherwise, leave the wagons on “one” and “two” unchanged.

  This adds one more wagon in the right position on “main”.
- Do the same for “two”.

Each recursive application of `FewerFind` has of course to supply the wagons on all three tracks correctly!

Initially, `FewerFind` is applied such that the tracks “one” and “two” are the empty list. For example,

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

returns

```
[one(1) two(1) one(~1) two(0) one(0) two(~1)]
```

Acknowledgments

The idea and the initial problem formulation is taken from an assignment at the 8th Prolog Programming Competetion organized by Bart Demoen and Phuong-Lan Nguyen. The current formulation is due to Christian Schulte.