2012 Haskell January Test
Labelled Transition Systems

This test comprises four parts and the maximum mark is 30. Parts I–III are worth 28 of the 30 marks available. Part IV carries 4 marks, giving a potential total of 32 marks, but your final mark will be capped at 30. The **2012 Haskell Programming Prize** will be awarded for the best attempt(s) at Part IV.

Credit will be awarded throughout for clarity, conciseness, *useful* commenting and the appropriate use of Haskell's various language features and predefined functions.

WARNING: The examples and test cases here are not guaranteed to exercise all aspects of your code. You may therefore wish to define your own tests to complement the ones provided.

1 Introduction

State Transition Systems are used extensively in computing to model the behaviour of a system at an abstract level and, subsequently, to establish whether it behaves as intended, e.g. with respect to given correctness and/or performance criteria. This exercise focuses on a class of transition systems called *Labelled Transition Systems* (LTSs) and an accompanying 'language' called FSP (an acronym for *Finite State Process*) that can be used to define them. This exercise, including many of the examples used, is based on the material in the popular textbook:

> "Concurrency: State Models and Java Programs" by J. Magee and J. Kramer, 2nd Edition, John Wiley & Sons, ISBN:978-0-470-09355-9.

2 Labelled Transition Systems (LTSs)

An LTS is a directed graph where the nodes of the graph correspond to the *states* that a system can be in and where the edges correspond to *transitions* which model *actions* that change the system's state. Each transition is labelled with an *action name* which, by convention, is a string beginning with a lower case letter. We say that a transition from state s to state t occurs as the result of 'executing' an action a if there is a transition from s to t labelled with a in the LTS. The resulting transition will be written $s \rightarrow_{\mathsf{a}} t$. Here, s is the *source* state and t the *target* state of the transition.

Throughout this exercise each state is assumed to be numbered using a non-negative integer identifier and there will always be a *starting state* that has identifier 0; the other states (if there are any) can be labelled with arbitrary positive integers, so long as they are unique. In a given state there may be more than one possible transition, in which case any one of the corresponding actions can execute. In this case we say that there is a *non-deterministic choice* among the set of actions at that point.

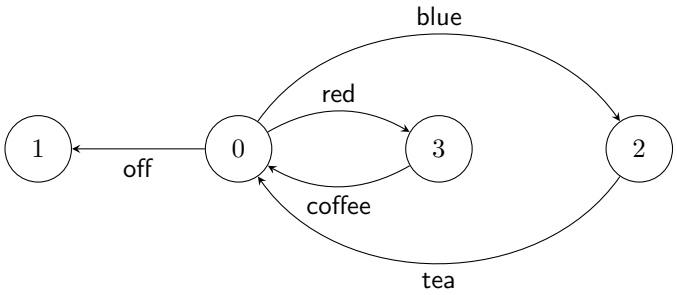


Figure 1: The LTS for the vending machine.

Figure 1 shows an example LTS that represents, at an abstract level, the behaviour of a simple vending machine that dispenses coffee or tea after a red or blue button is pressed, or turns off (forever) after an off switch is pressed. There are four states numbered 0, 1, 2 and 3, and five transitions. The three transitions out of state 0 (labelled off, blue and red) correspond to the non-deterministic choice between the three buttons that can be pressed when the machine is in state 0. The set of action names in an LTS is called the *alphabet* of the LTS; in the example the alphabet is {red, blue, tea, coffee, off}.

In this exercise, an LTS will be represented in Haskell by a list of transitions, thus:

```
type Id         = String
type State      = Int
type Transition = ((State, State), Id)
type LTS        = [Transition]
type Alphabet   = [Id]
```

For example, the LTS of Figure 1 can be represented by the list:

```
[((0,1),"off"),((0,2),"blue"),((0,3),"red"),((2,0),"tea"),((3,0),"coffee")]
```

Note that the order in which the transitions are listed is unimportant.

All LTSs can be assumed to be *well-formed* in the sense that every state in the LTS is *reachable* from state 0, i.e. there is a (possibly empty) sequence of transitions, i.e. a *path*, from state 0 to every other state.

2.1 Traces

An LTS defines a potentially infinite set of *execution traces* which are obtained by following a sequence of transitions from the starting state, 0. For example, the LTS for the vending machine in Figure 1 includes the traces:

blue → tea → off → ...
blue → tea → red → coffee → ...
red → coffee → blue → tea → ...
red → coffee → red → coffee → ...
...

and so on. Note that in general an LTS may have infinitely many traces and that some traces may be infinitely long by virtue of cycles in the LTS. Conversely, if an LTS contains no cycles then it will have a finite number of traces, each of which will be finite in length.

At this point you are in a position to answer the questions in Part I.

3 Finite State Processes (FSP)

FSP is a simple recursive language for specifying labelled transition systems at a high level. An FSP program comprises a set of process definitions where, by convention, each process name is an upper case string, as in `NAME =` ... where the "..." denotes a *process*. An FSP process is either:

1. `STOP`, the 'stopped' process, for which there are no executable actions.
2. A *reference* to a named process in the program, i.e. an upper case string, for example `P`, `Q`, ...
3. An *action prefix* of the form `a -> P`, where `a` is an *action label* (a string beginning with a lower case letter) and `P` is a process.
4. A *choice* of the form `(P1 | P2 | ... | Pn)` where each `Pi` is a process. Choice processes can be assumed to be well formed in the sense that each `Pi` will begin with an action prefix; the choice thus defines a *non-deterministic* choice between the execution of the actions of each subprocess.

Note that a reference can be replaced by its definition without changing the meaning of the process which uses it, e.g. the recursive definition `P = a -> P` has the same meaning as `a -> a -> P` and also `a -> a -> a -> P` and so on.

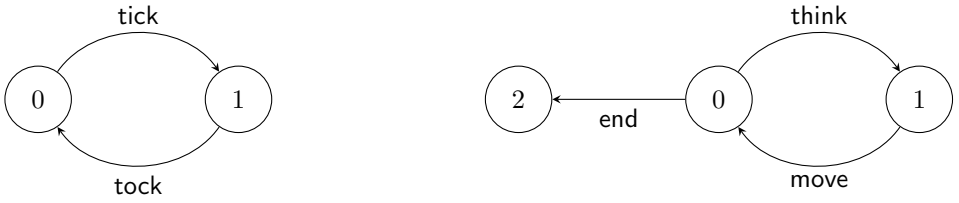
As an example, the following FSP process corresponds to the vendor LTS shown in Figure 1.

```
VENDOR = (red -> coffee -> VENDOR |
          blue -> tea -> VENDOR |
          off -> STOP)
```

Note that the cycles in the LTS arise as a result of the recursive references to `VENDOR` inside its own definition.

In the same way that an LTS can generate a set of traces, so can an FSP process, this time by following action prefixes and replacing process names, e.g. "`VENDOR`", by their defining process. As an example, the `VENDOR` process above produces the identical set of traces to its corresponding LTS (Figure 1), some examples of which were listed in Section 2.1 above.

Two further examples of FSP processes are `CLOCK` and `PLAY`, whose definitions and corresponding LTSs are shown in Figure 2.




```
CLOCK = (tick -> tock -> CLOCK)
```

```
PLAY = (think -> move -> PLAY | end -> STOP)
```

Figure 2: The `CLOCK` and `PLAY` processes.

A process corresponds to an LTS in the sense that each syntactic (sub)term that can be reached following the execution of zero or more actions defines a unique state that the process can be in. Furthermore, each action prefix corresponds to a transition from one state (subterm) to another. For example, in the `PLAY` process of Figure 2, `PLAY` itself (the start process) corresponds to state 0 in the LTS and the two subterms '`move -> PLAY`' and '`STOP`' (reached after execution of `think` and `end` respectively) to states 1 and 2 respectively. The (recursive) reference to `PLAY` is synonymous with the start process, and hence state 0. The transitions in the LTS correspond to the action prefixes '`think ->`...' (state 0 to state 1), '`move -> PLAY`' (state 1 to state 0) and '`end -> STOP`' (state 0 to state 2). Because the `STOP` process has no executable actions there are no transitions from state 2.

FSP process definitions can be represented in Haskell by the following types:

```
data Process  = STOP |
                Ref Id |
                Prefix Id Process |
                Choice [Process]
              deriving (Eq,Ord,Show)

type ProcessDef = (Id, Process)
```

As an example, the `VENDOR` process can be represented as follows:

```
vendor = ("VENDOR",
          Choice [Prefix "red"  (Prefix "coffee" (Ref "VENDOR")),
                  Prefix "blue" (Prefix "tea" (Ref "VENDOR")),
                  Prefix "off" STOP])
```

The representations of all the FSP processes defined in this document are included in the template file.

At this point you are in a position to answer the questions in Parts I and II. As the next section is quite involved you may wish to complete these parts before reading on.

4 Composition

The *composition* of two FSP processes yields an LTS which is obtained by composing the LTSs of the individual processes. The resulting LTS captures all possible interleavings of the individual LTSs. As an example, the LTS for the composition of the `CLOCK` and `PLAY` processes in Figure 2 is shown in Figure 3.

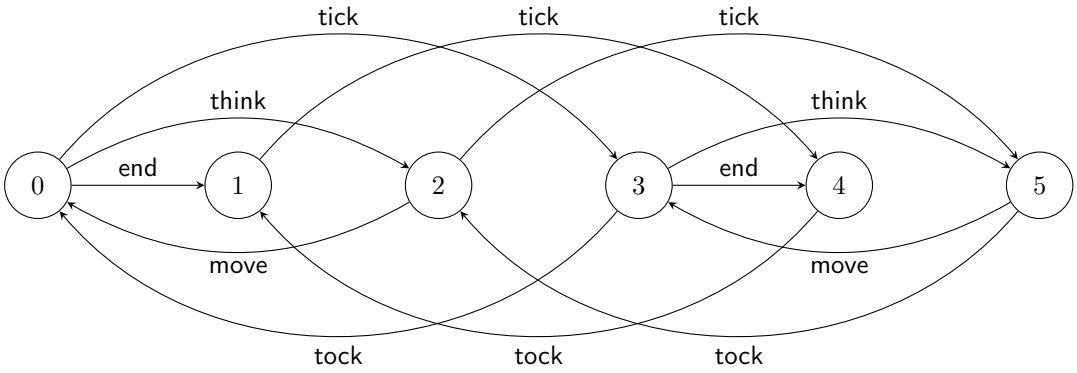


Figure 3: The LTS for the composition of `CLOCK` and `PLAY`.

The six states, numbered 0–5, in the composed process correspond to the *pairs* of states $(0, 0)$, $(0, 2)$, $(0, 1)$, $(1, 0)$, $(1, 2)$, and $(1, 1)$ respectively, where, in the pair (s, t), s is a state of the `CLOCK` process and t is a state of the `PLAY` process. The following are thus valid traces in the composed LTS:

$\mathsf{end} \rightarrow \mathsf{tick} \rightarrow \mathsf{tock} \rightarrow \ldots$
$\mathsf{tick} \rightarrow \mathsf{end} \rightarrow \mathsf{tock} \rightarrow \ldots$
$\mathsf{tick} \rightarrow \mathsf{tock} \rightarrow \mathsf{tick} \rightarrow \mathsf{think} \rightarrow \ldots$
$\mathsf{think} \rightarrow \mathsf{tick} \rightarrow \mathsf{move} \rightarrow \mathsf{end} \rightarrow \ldots$
$\mathsf{think} \rightarrow \mathsf{tick} \rightarrow \mathsf{tock} \rightarrow \mathsf{move} \rightarrow \ldots$
$\ldots$

and so on.

4.1 Synchronization

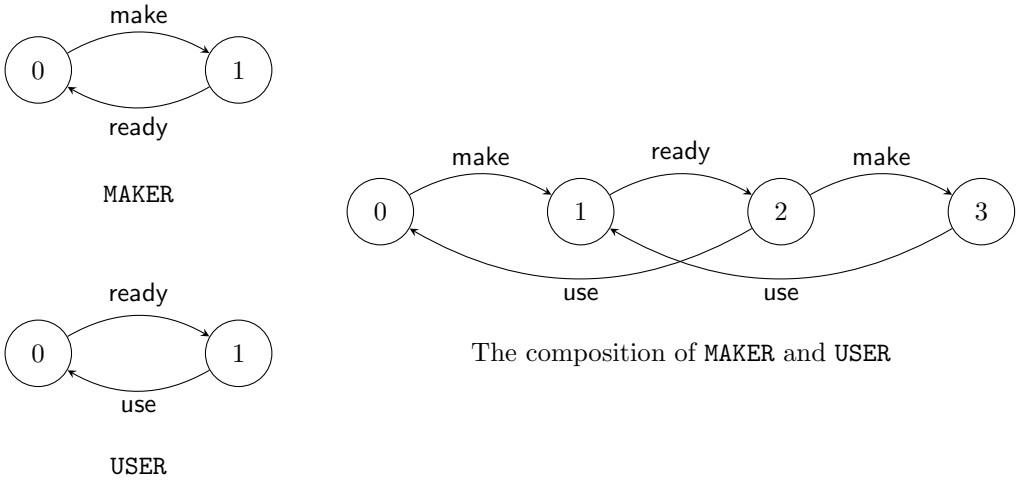
If two processes have an action in common then the shared action must be executed at the same time by both processes – this models *synchronization*. As an example, consider the processes:

```
MAKER = (make -> ready -> MAKER)
USER  = (ready -> use -> USER)
```

Here the processes have a shared action, `ready`, i.e. the action label `ready` is a member of the alphabets of both processes. Thus, a `ready` action in one process cannot execute until a `ready` action can execute in the other. The LTS for the composition of `MAKER` and `USER` is thus as shown in Figure 4, and some valid traces include:

$\mathsf{make} \rightarrow \mathsf{ready} \rightarrow \mathsf{use} \rightarrow \mathsf{make} \rightarrow \mathsf{ready} \rightarrow \mathsf{make} \rightarrow \ldots$
$\mathsf{make} \rightarrow \mathsf{ready} \rightarrow \mathsf{make} \rightarrow \mathsf{use} \rightarrow \mathsf{ready} \rightarrow \mathsf{use} \rightarrow \ldots$
$\ldots$

Note that because the actions `make` and `use` are not shared they can execute without constraint. Indeed, if both `make` and `use` are available for execution at the same time, as in state 2, then either can execute first (non-deterministic choice).



The composition of `MAKER` and `USER`

Figure 4: The `MAKER` and `USER` processes, together with their composition.

5 Implementing Composition

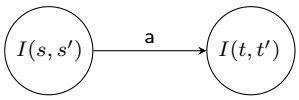
The composition of two LTSs is built by composing pairs of transitions in the individual LTSs and then pruning the resulting list of transitions so that only those reachable from state 0 are included; the resulting list of transitions is then a well-formed LTS.

5.1 Composing Two Transitions

The composition of two transitions will generate either zero, one or two *new* transitions which may ultimately form part of the required LTS. Suppose the two transitions in question are $s \rightarrow_{\mathsf{a}} t$ (from the first LTS) and $s' \rightarrow_{\mathsf{a}'} t'$ (from the second LTS). Each new transition, if there is one, will have a source identifier that corresponds to the *pair* of states (s, s') and a target state identifier that corresponds to a pair that is some combination of the states s, s', t and t'. In what follows, the mapping from these state pairs to new identifiers is assumed to be defined by a function I. For example, for the composition of `CLOCK` and `PLAY` above, I will be such that $I(0,0)$=0, $I(0,2)$=1, and so on (see Section 4). We assume that it will always be the case that $s_1 = s'_1 = 0$ and that $I(s_1, s'_1) = 0$, i.e. state 0 in the composed process always corresponds to the pair comprising the 0 states in the constituent LTSs.

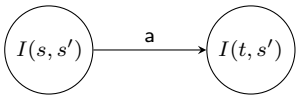
The composition of the two transitions $s \rightarrow_{\mathsf{a}} t$ and $s' \rightarrow_{\mathsf{a}'} t'$ is built according to the following rules. These rules *must* be read *in order*, as if they were a sequence of Haskell guards (that's a hint!); recall that α_1 and α_2 denote the alphabets of LTS_1 and LTS_2 respectively:

1. If $\mathsf{a} = \mathsf{a}'$ then the action is shared and the result is a single new transition from state pair (s, s') to (t, t') labelled a, i.e. the new transition is $I(s, s') \rightarrow_{\mathsf{a}} I(t, t')$, using the new state numbering scheme. Otherwise, ...

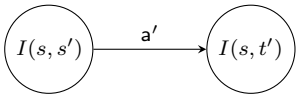


2. If $\mathsf{a} \in \alpha_2$ and $\mathsf{a}' \in \alpha_1$ then neither a nor a' can execute so there are no resulting transitions. Otherwise, ...

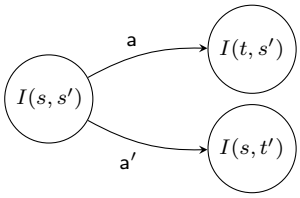
3. If $\mathsf{a}' \in \alpha_1$ then a' cannot execute but a can, so the result is a single new transition $I(s, s') \rightarrow_{\mathsf{a}} I(t, s')$. Otherwise, ...



4. If $\mathsf{a} \in \alpha_2$ then a cannot execute but a' can, so the result is a single new transition $I(s, s') \rightarrow_{\mathsf{a}'} I(s, t')$. Otherwise, ...



5. Both a and a' can execute without constraint so the result comprises the *two* new transitions: $I(s, s') \rightarrow_{\mathsf{a}} I(t, s')$ and $I(s, s') \rightarrow_{\mathsf{a}'} I(s, t')$.



You are now in a position to answer the questions in Part III. You might wish to complete these before reading Section 5.2.

5.2 Composing LTSs

In the algorithm you are required to implement, the composition of two LTSs is produced by first building the *cartesian product* of the sets of states in each LTS. Suppose the two LTSs are called LTS_1 and LTS_2 with alphabets α_1 and α_2 respectively. Suppose also that they comprise the states $s_1, s_2, \ldots, s_m$ and $s'_1, s'_2, \ldots, s'_n$ respectively, for some $m, n > 0$. From Section 5.1 we also have that $s_1 = s'_1 = 0$. The composed LTS will comprise the new states $0, 1, \ldots, mn-1$ corresponding to all possible *pairs* of states in the constituent processes, i.e. the cartesian product: (s_1, s'_1), (s_1, s'_2), …, (s_2, s'_1), (s_2, s'_2), …, (s_m, s'_n). The I function above encodes this mapping where, as noted, $s_1 = 0$, $s'_1 = 0$ and $I(s_1, s'_1) = 0$.

The next task is to determine the transitions between these new states. To do this, look at each pair of states in turn. For each such pair, (s, s') say, you first need to extract the sets of transitions out of the two states in the original LTSs. Suppose state s in LTS_1 has transitions to target states $t_1, \ldots, t_u$ for some integer u with corresponding action labels $\mathsf{a_1}, \ldots, \mathsf{a_u}$; similarly $t'_1, \ldots, t'_v$ for state s' in LTS_2 with corresponding action labels $\mathsf{a'_1}, \ldots, \mathsf{a'_v}$. This situation is depicted in Figure 5. The algorithm now proceeds by inspecting all possible pairs of transitions from these two sets (another cartesian product) and applying the rules given in Section 5.1 above to compose them. The final list of transitions is the concatenation of the results of the individual transition compositions.

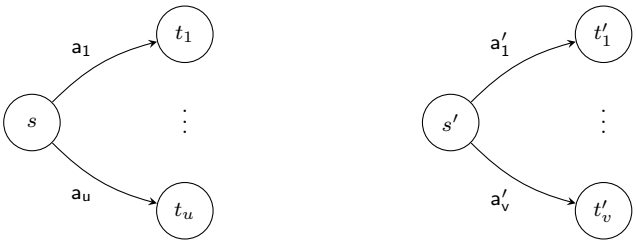


Figure 5: Composing a pair of states requires examining the cartesian product of their transitions. s is a state of LTS_1 and s' a state of LTS_2.

5.2.1 Pruning

A feature of the algorithm above is that it may generate more states, and thus more transitions, than are required to build a well-formed LTS. For example, composing the LTSs for the two processes

```
P = a -> b -> c -> STOP
Q = d -> c -> b -> Q
```

yields an LTS with just four states (Figure 6), even though the LTSs for `P` and `Q` comprise 4 and 3 states respectively, suggesting a composed LTS with 12 states. The reason is that for the two subprocesses `b -> c -> STOP` and `c -> b -> Q` no action is executable, as they share the actions `b` and `c`; only actions `a` and `d` can execute without constraint. One way to fix this is to 'prune' the resulting states to include only those reachable from state 0. Figure 6 shows the result of this process, where the states other than 0, 1, 3 and 4 have been pruned.

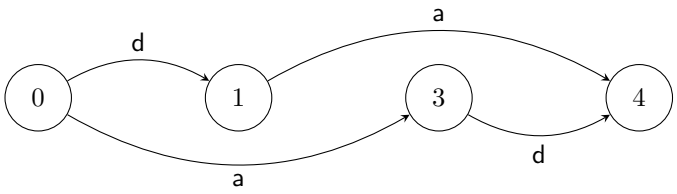


Figure 6: The LTS for the composition of `P` and `Q`.

5.2.2 The Catch

There is one final problem: if one of the states being considered at any point has *no* outgoing transitions then the transitions of the other state will be ignored, since the cartesian product of the two sets of transitions will be empty. Suppose state s has no outgoing transitions and state s' has zero or more transitions labelled $a_1, a_1, ..., a_n$, for some $n \geq 0$. One way to fix the problem in this case is to compose the transition $s' \rightarrow_{\mathsf{a_i}} t_i$ from s to state t_i, say, with an artificially added 'self' transition $s \rightarrow_{\mathsf{a_i}} s$, $0 \leq i \leq n$. The rules for transition composition (Section 5.1) will then ensure that a single new transition $I(s, s') \rightarrow_{\mathsf{a_i}} I(s, t_i)$ is created in the composed LTS, as required.

6 What To Do

The questions are spread over four parts. If you get stuck at any point you are advised to try a different question, as most of the questions can be answered independently of the others.

A number of test FSP process definitions are included in the template, along with their corresponding LTSs that you can use for testing purposes. By convention, the LTS for the FSP process represented in Haskell by `p` is named `pLTS`.

6.1 Part I – LTS Utilities

1. Define a function `lookUp :: Eq a => a -> [(a, b)] -> b` that will look up a given item in a list of (item, value) pairs. A precondition is that the given item is present in the list.

   [1 mark]

2. Using `nub`, define a function `states :: LTS -> [State]` that computes a list of the state indices in a given LTS, without duplicates. For example, `states vendorLTS` should return `[0,1,2,3]` in some order. Hint: `nub` *once* by defining a helper function and `nub`bing the list it returns.

   [2 marks]

3. Define a function `transitions :: State -> LTS -> [Transition]` that will return the list of transitions from a specified state in a given LTS. For example, `transitions 0 vendorLTS` should return the list `[((0,1),"off"),((0,2),"blue"),((0,3),"red")]` in some order.

   [3 marks]

4. Define a function `alphabet :: LTS -> Alphabet` that will return the alphabet of a given LTS, again with no duplicates. For example, `alphabet playLTS` should return the list `["end","move","think"]` in some order.

[2 marks]

6.2 Part II – FSP Functions

1. Define a function `actions :: Process -> [Id]` that will compute the (duplicate free) list of action names in a given FSP process. For example, `actions (snd maker)` should return `["make","ready"]` in some order. Recall that `maker` is of type `ProcessDef`, hence the use of `snd` to extract its corresponding `Process`. Note that a testable axiom is that `actions p` should contain the same elements as `alphabet pLTS`. Hint: both the `STOP` and `Ref` constructors yield no actions (`[]`).

   [5 marks]

2. Define a function `accepts :: [Id] -> [ProcessDef] -> Bool` that delivers `True` iff the given list of process definitions is capable of producing the given trace (list of action identifiers), which can be assumed to be finite. A precondition is that the first item in the list of process definitions is that of the start process. For example, `accepts ["on","off","on"] [switch,on,off]` should return `True`; `accepts ["use","use"] [user]` should return `False` and `accepts [] [user]` should return `True` (every process accepts the empty trace). Hint: Use a helper function to recurse over the start process. Note also that the `Ref` constructor case in the helper involves a look-up in the given list of definitions.

   [6 marks]

6.3 Part III – Transition Functions

1. Define a *non-recursive* function `composeTransitions` that implements transition composition as described in Section 5.1. The function's type is:

```
composeTransitions :: Transition -> Transition
                   -> Alphabet -> Alphabet
                   -> StateMap
                   -> [Transition]
```

   The first transition is assumed to come from an LTS whose alphabet is given by the first `Alphabet` parameter; likewise the second. The `StateMap` is a table that encodes the I function described in Section 5.1:

```
type StateMap = [((State, State), State)]
```

   Thus, if `m` is a state map then `lookUp (s, s') m` will compute $I(s, s')$. All (four) possible pairs of source and target states drawn from the two argument `Transitions` can be assumed to be contained in the given `StateMap`. For example, if `m` is a state map given by:

```
m = [((0,0),0),((0,1),1),((1,0),2),((1,1),3)]
```

   then:

```
*Main> composeTransitions ((0,1),"a") ((0,1),"c") ["a"] ["b","c"] m
[((0,1),"c"),((0,2),"a")]
*Main> composeTransitions ((1,1),"a") ((1,0),"c") ["a"] ["a","c"] m
[((3,2),"c")]
*Main> composeTransitions ((0,1),"a") ((1,1),"c") ["a","c"] ["a","c"] m
[]
```

Note that the resulting list of transitions comprises either 0, 1 or 2 elements.

[5 marks]

2. Define a function `pruneTransitions :: [Transition] -> LTS` that, given a list of transitions, will return those that are 'reachable' from state 0, i.e. those transitions that collectively form the paths from state 0 to the states reachable from it. For example, `pruneTransitions [((0,1),"d"),((1,4),"a"),((2,5),"a")]` should return `[((0,1),"d"),((1,4),"a")]`, as states 2 and 5 are not reachable from state 0.

   Hint: Use a helper function `visit :: State -> [State] -> [Transition]` that accepts a state to be visited (initially 0) and a list of states that have already been visited (initially `[]`). When visiting a state s, say, that has *not* been visited before `visit` should process each of the outgoing transitions of state s, obtained using the `transitions` function above. Each such transition, `((from, to), a)`, say, forms part of the solution and the search should then continue by visiting the target state (`to`), noting that `from` should be added to the list of seen states in the recursive call.

[4 marks]

6.4 Part IV – LTS Composition

1. Using `composeTransitions` and `pruneTransitions` from Part III above, define a function `compose :: LTS -> LTS -> LTS` that composes two labelled transition systems using the method described in Section 5.

[4 marks]

6.5 Part V – LTS Construction

Welcome to the hidden part(!), for which there are no marks. Anyone completing this question will be inducted into Imperial College's **Haskell Hall of Fame**. Good luck!

1. Define a function `buildLTS :: [ProcessDef] -> LTS` that will build an LTS for the start process in the given list of FSP process definitions. Recall that the start process is always at the head of the list of definitions. A precondition is that if the start process is defined in terms of one or more additional processes then their definitions will also be contained in the list of definitions. Thus, for example:

```
*Main> buildLTS [vendor]
[((0,3),"red"),((3,0),"coffee"),((0,2),"blue"),((2,0),"tea"),((0,1),"off")]
```

Remember that the start state must be numbered 0, but that the remaining state indices can be arbitrary positive integers so long as the LTS (graph) structure is correct. Note also that there may be many (possibly infinitely many) LTSs which are behaviourally identical, in that they produce identical sets of traces. For example, the `SWITCH` process given by:

```
SWITCH = OFF
OFF    = (on -> ON)
ON     = (off -> OFF)
```

has an optimal LTS with two states, e.g. `[((0,1),"on"),((1,0),"off")]`, but the LTS `[((0,1),"off"),((1,2),"on"),((2,1),"off")]`, which has three states, also produces the same traces. For this exercise, the less optimal LTS is OK – you'll still be inducted into the Hall of Fame!

Hint: There are many ways to solve the problem, but you might like to consider writing a helper function that carries, among other things, the state identifier to be assigned to that process and the next available identifier (for use in recursive calls, presumably). What should it return? The LTS for the process and next available identifier (of course), but what else...? It may also be useful to note that `Process` is an instance of `Eq`.

[0 marks, but eternal glory]