Tutorial 3: Payments, tokens, and other credentials

With the first two tutorials complete, all the basic elements of Fae necessary to construct an economy have been presented. However, as we saw at the end of Tutorial 2, assembling them into a secure interface that protects value requires the application of idioms that transcend their coding elements. There, we saw the idiom for encapsulating a valuable as a controlled interface to a module containing its implementation, similar to the object-oriented pattern of coding. Values defined in this way are secure "credentials", in that possession of them is special, and in this tutorial, we will illustrate how they can be used in simple and complex contract logic to differentiate the experiences of various Fae users.

This tutorial will walk through constructing a kind of lottery based on `Nametag`s. This simple "valuable" we created in the last tutorial does not have a lot of features but can still serve as an identifier in a game where names are "drawn from a hat". This toy example serves as a proxy for (aspects of) any number of serious applications, such as membership-based clubs, actual lotteries, voting, opinion polls, and so on. Unlike most of our previous examples, it is not transaction-focused: the main activity is to define a contract that runs the lottery, while any transactions will consist almost entirely of calling this contract and others. Their activities with the results are uninteresting.

As in previous tutorials, the sample code will use the conventions of the postTX utility and the sample command lines use the docker images. After following the instructions there, you should execute this first to have a local server running:

```
./faeServer.sh
```

Lottery state

The first way that this contract differs from all previous ones is that it maintains a nontrivial *state*. The closest we have yet come was in an early example where the contract's first call first called `release` and its second call called `spend`; this variable activity depending on the contract's history is a kind of state, but no user-generated data was stored. For the lottery, we will want to store the details of which names are in the hat, how many more will be accepted, and, when the lottery has ended, who won.

To keep the rules simple, we will say that the lottery is nonrandom, accepts a specific number of names, and splits the award among all the users (identified by their public key identities) who submitted the maximum number of names. The state type we will use is:

Snippet: Lottery state type

```
data LotteryState =
  RunningState
  {
    nametags :: Entries,
    count :: Natural,
    winningCount :: Natural
  } |
  FinishedState
  {
    winners :: Entries,
    totalWinners :: Natural,
```

```
    nonWinners :: Entries
  }
```

The obviously important `Entries` type is supposed to hold the various submissions. We will track submissions by the public key of the user who made them, and store all the nametags that they have submitted to be returned later. The appropriate type for this is:

```
type Entries = Map PublicKey (Seq Nametag)
```

In words, a mapping that, for each public key, gives the sequence of nametags that key has entered.

SUM TYPES

Unlike the other types we have defined previously, `LotteryState` has multiple constructors (single constructors were introduced in the discussion of contract names). In other words, there are two kinds of states, whose natures are described by the constructor names `RunningState` and `FinishedState`. Each kind of state has its own data, completely independent of the others.

MAP AND SEQ

The two containers present in `Entries` are new to us. These are defined in the standard libraries, respectively, `Data.Map.Strict` and `Data.Seq`. There is actually a `Data.Map.Lazy` as well, the only difference in which is that it defers evaluation of the map keys (the various `PublicKeys`) until they are specifically examined; we use the strict version because it helps us ensure that runtime errors happen when they are caused, not simply when they are noticed.

A `Map` is a two-parameter polymorphic type (we saw one-parameter polymorphism in the last tutorial), `Map keyType valType`, where the parameters describe exactly the types that index the map and that reside in it. In order to use any of its API functions, we require the constraint `(Ord keyType)` (constraints were discussed alongside polymorphism), where `Ord` is the typeclass (interface) that describes types that support the comparison operators `<` and `>` (as well as the equality operator `==` from the `Eq` typeclass; `Ord` extends `Eq`), because the map is actually a balanced search tree internally.

A `Seq` is a one-parameter polymorphic type, `Seq valType`, similar to the list type `[valType]` except more like an array. We use it because it has a constant-time `length` function, which lists do not.

NATURAL

The `Natural` type is an unbounded positive integer type. Using it is something of a conceit here, since the actual entries cannot occupy an unbounded amount of memory, but it reflects the fact that the bit-size bounds on built-in integer types have no relation to the other details of the lottery; likewise, the signedness of `Integer` is unnecessary. Using `Natural` allows the interpreter to catch errors arising from the inappropriate use of these numbers, should we make any.

HOW IT WORKS

This declaration does not “do” anything, but it does describe our intentions for the lottery.

The `RunningState` variant has its definition because:

- It records the nametags that have been entered, sorted by entrant.
- It also records the total number of entries, so that we can easily compare it to the lottery limit. (We don't technically need this, as it is implicit in `nametags`, but to extract it requires traversing the entire map and summing all the lengths of the various sequences, every time a new entry is made, which is pointless.)

- It also records the *winning* number of entries, which will always be the maximum number for any entrant and may occur multiple times. This, likewise, is a minor optimization that replaces a traversal of the map to find the longest sequence.

The `FinishedState` variant has its definition because:

- It records the winning and non-winning participants separately. This determination can only be made after the last entry has been made, since the winners are constantly evolving until then. Of course, winners and losers get a different response from the lottery contract, so they need to be distinguished. This is, again, an optimization replacing the repeated sorting of the `nametags` mapping each time a response is made.
- It also records the *total number* of winners. This information is actually useless to the lottery contract itself but may be useful to the participants (say, to justify splitting a big monetary award) and cannot be derived from any other information they would receive.

All of the optimizations mentioned here are more oriented towards eliminating excessive and complex code than saving time or space during execution.

Lottery contract name

As a contract, the lottery needs a contract name, first described in Tutorial 1. This name reflects the parameters of the lottery and is also associated with the argument and value types of the contract function.

Snippet: Contract name, argument, and value types

```
data Lottery =
  Lottery
  {
    limit :: Natural,
    owner :: PublicKey
  }
  deriving (Generic)

data LotteryAction = Enter | Exit deriving (Generic, Read)

data LotteryResult =
  EnterResult
  {
    enterCount :: !Natural,
    message :: !String
  } |
  ExitResult
  {
    lotteryLimit :: Natural,
    returned :: !Nametags
  } |
  WinResult
  {
    winnersCount :: Natural,
    returned :: !Nametags
  }
  deriving (Generic)

type Nametags = Container (Seq Nametag)
```

LOTTERY NAME

The contract name for the lottery will be, of course, the `Lottery` type, which actually only tracks a single `Natural` number, the maximum number of entries before the lottery closes.

MODAL ARGUMENTS

The `LotteryAction` (which will be the `ArgType Lottery`) is *modal*, meaning that it contains no parameters (both constructors are *nullary*) and simply flags a particular mode of operation. The reason for this pattern will be explained when we get into the contract code.

THE READ CONSTRAINT

`LotteryAction` derives a typeclass `Read`, which is necessary for the argument types of contracts (but not escrows) because their arguments are specified as text literals in the transaction message and need to be parsed into their correct representations when the contract call occurs. This can be derived for any type that consists of "values" (i.e. not functions), but occasionally, it is useful to take the reins and write your own instance. Either way, the translation of literal arguments into abstract types is entirely under the control of the contract author.

BANG PATTERNS

The `!` annotation on most of the fields of `LotteryResult` serve the purpose of marking "things which changed" during the course of a single call to the lottery contract, and therefore, which need to be "forced", i.e. evaluated now rather than deferred until some later time when they are forced. This is important for isolating the effects of a contract call to that call, rather than allowing errors to sneak out and only be reported in the transaction body or (even worse) later calls to the contract.

THE CONTAINER TYPE

The definition of `Nametags` includes a type constructor we have not yet seen, `Container`. This tags the `Seq Nametag` as being a *container of value-bearing objects*, and indicates to Fae to introspect into that container. This facility is, as mentioned in that discussion, automatic for types deriving `Generic`, but not all types have a meaningful generic structure; for instance, as mentioned above, the `Map` type is actually a balanced tree internally, and its implementation involves hidden fields that may also not cooperate with Fae. However, the common feature of containers is that their values can be *traversed*, and most of the library-provided containers do have an instance of the typeclass `Traversable` that describes this, and which Fae uses to get the escrows backing each object in the container.

(The `Set` type is the exception to this rule, because it requires its value types to be `Ord` instances, which for not-necessarily-good technical reasons precludes an instance of `Traversable`).

In short, any type that functions as a container should be tagged with `Container` wherever it will be transferred between contracts, such as in the return value type here.

HOW IT WORKS

The `LotteryAction` is self-explanatory, except that the `Enter` variant mysteriously contains no information about a `Nametag` to be entered. This will be handled by a feature to be described in the code.

The `LotteryResult` has *three* constructors, one of which is to be returned in response to `Enter` and the other two of which, in response to `Exit`. Their rationales are:

- `EnterResult` returns the running count of submissions for the entrant who made the call, and the message produced by the nametag they submitted. Neither one of these is truly unobtainable otherwise, but the former

is informative and the latter is a convenient excuse to force the evaluation (and hence verification) of the nametag.

- `ExitResult` returns the entrant's nametags along with the information of the lottery limit, with the interpretation “you submitted these entries out of so many allowed”.
- `WinResult` is delivered to winners after the lottery ends, and contains in addition to their nametags the number of winners (at least 1, but possibly any number).

Exceptions

Deviating slightly from the thread of the lottery's implementation, we take a moment to define an exception type for errors that may occur in the contract.

Snippet: Lottery exception type

```
data LotteryError = NotAuthorized | Finished | NotFound | BothMaps

instance Show LotteryError where
  show NotAuthorized = "This action was not authorized by the lottery owner"
  show Finished = "The lottery is finished; no new entries accepted"
  show NotFound = "Not an entrant or already exited"
  show BothMaps = "Internal error; entrant is in both maps"

instance Exception LotteryError
```

ERROR MESSAGES

There are four kinds of error conditions in the lottery, whose functions are mostly adequately described by the error text in their `Show` instances:

- `NotAuthorized`, meaning that the owner of the lottery has not “signed off” on the submission. This is crucial because the contract deals in unsafe values, namely, `Nametag`s of unknown origin that may be invalid or even nonterminating values. The lottery must validate the submissions, but in the latter case, to do so would cause the present call to hang, and therefore, shut down the contract for all future calls. It is the lottery owner's responsibility to curate the contract to prevent this (by, for instance, simulating the call in a sandbox). Other parties, of course, may not do so.
- `Finished`, meaning that an action (submitting a new entry) was requested that can only be done when the lottery is ongoing, but it is actually no longer ongoing.
- `NotFound`, meaning that someone tried to reclaim their entry (if it lost) or their winnings, and there is no record of them, for one of the two reasons given.
- `BothMaps`. This error only exists because it is implied by a particular branch of the control flow that, if the contract is well-written, should not happen, but of course, may anyway. It is a failure of the handling of state.

EXCEPTION

The `LotteryError` type is made an instance of the `Exception` class so that it can be `thrown` when errors arise. This simple declaration requires only that it already be an instance of `Show`, making a generic `Exception` no more than a well-typed string, though they can in fact have data not reflected in the string that may be interesting when the exception is received by `catch`, a function that Fae disables; exceptions in a contract call will be reflected in the transaction's return value and can be caught by the observer that evaluates this.

Lottery contract function

Having introduced all the types that enter into the lottery, we can now begin to implement it by giving a `ContractName` instance to `Lottery`:

Snippet: Lottery's instance of ContractName

```
instance ContractName Lottery where
  type ArgType Lottery = LotteryAction
  type ValType Lottery = LotteryResult
  theContract Lottery{..} = usingState startingState $ feedback $ \case
    Enter -> newEntry limit owner
    Exit -> getEntry limit

startingState :: LotteryState
startingState = RunningState Map.empty 0 0
```

As promised earlier, the lottery function will accept a `LotteryAction` and return a `LotteryResult`.

RECORD WILDCARDS

The `theContract` function accepts a `Lottery` value that directs the actions of the lottery. The pattern syntax `Lottery{..}` is a *record wildcard* that simply binds each of the field names of a `Lottery` (namely, `limit` and `owner`) to their corresponding values in the scope of the function.

LAMBDA CASE

The `\case` syntax is a *lambda case* expression, a shorthand for the common but verbose idiom

```
\x -> case x of ...
```

where the cases appear below, namely, the `Enter` and `Exit` patterns exhausting a `LotteryAction`'s possible values.

USINGSTATE

The `usingState` function is not precisely an API function but rather a helper for defining contracts with state. Its first argument is an initial value for the state, which here is of type `LotteryState` and contains a lot of nothing in the context of a lottery that has not finished. Its second argument is a function with signature

```
LotteryAction -> StateT LotteryState (Fae LotteryAction LotteryResult) a
```

(for some `a`) that resides in a *monad stack* atop the base `Fae` monad. The `StateT` *monad transformer*, defined in `Control.Monad.Trans.State`, adds to its base monad a small API that allows getting and setting a `LotteryState` that is passed along through its execution. The purpose of `usingState` is to initialize this passed-along value and bring the function down to the correct signature `LotteryAction -> Fae LotteryAction LotteryResult a` required by `theContract`.

FEEDBACK

The `feedback` function is a particularly abstract helper that substantially changes the coding style required for contracts that, like a `Lottery`, are essentially infinite loops. At a high level, what it does is to take a function that handles the argument and returns a value (representing a single iteration of the loop, i.e. a single contract call) and to turn it into a function that has (nearly) the same signature but that `release`s that return value and feeds the result, the next call's argument, back to the original function.

A short example is the function

```
feedback $ \case
  True -> return 42
  False -> halt 57
```

where the `\case` is a non-iterative function that simply returns one of two `Integers` in response to a boolean argument, but with `feedback`, becomes a looping contract equivalent to

```
f = \case
  True -> do
    next <- release 42
    f next
  False -> spend 57
```

I, at least, believe that it is much more understandable with `feedback` than "naturally". A comparison of the two reveals that the `halt` function that was snuck in has the purpose of breaking the loop and terminating the contract, like `spend`.

The lambda case trails off with calls in each alternative of its argument to other functions, which will be discussed individually below.

New entries

When a lottery is ongoing, it may accept new submissions via the function:

Snippet: type signature of `newEntry`

```
newEntry ::
  Natural -> PublicKey ->
  StateT LotteryState (Fae LotteryAction LotteryResult) LotteryResult
```

Its arguments (as can be seen by comparison with the `theContract` definition above) are the lottery limit for entries and the identity of the lottery owner, and as promised, it operates in a state monad placed atop `Fae` (because of `usingState`) and returns a plain `LotteryResult` (because of `feedback`); this represents the action taken on a *single* new submission, without acknowledging the contract's need for iteration, which is handled by `feedback`.

The definition begins with a permissions check:

Snippet: checking the owner's signature

```
newEntry limit owner = do
  signedBy <- signer "owner"
  unless (signedBy == owner) $ throw NotAuthorized
```

As mentioned in the discussion of errors, the contract must be signed by the owner, and this enforces that requirement. This idiom has already been discussed in the context of restricted-access escrows. Note the use of `throw`, where previously we issued errors with a call to `error`.

Next, the function fetches the state and makes a decision based on it:

Snippet: taking action based on the state

```
  lotteryState <- get
  case lotteryState of
    FinishedState{} -> throw Finished
    RunningState{..} -> do
```

GET

The `get` function is one of the two that are part of the `StateT` API, and returns the `LotteryState` that is currently present as the ongoing state.

In this case, it is an error if the lottery is in a finished state, while if not, it proceeds to the main action:

Snippet: processing the entry

```
      nametag <- material "nametag"
      message <- checkNametag nametag
      entrant <- signer "self"
      let (enterCount, nametags') = addEntry nametag entrant nametags
          count' = count + 1
          winningCount'
            | enterCount > winningCount = enterCount
            | otherwise = winningCount
```

MATERIAL

We see a new Fae API function here, `material`. Syntactically, this functions much like `signer` and fetches a value by name; these values are passed in the transaction message, as we will see later. Unlike `signer`, the value may have any type and if there is a type mismatch with its usage then Fae throws an error. This mechanism is the only way to pass an escrow, or a value containing escrows, into a contract, since the actual arguments are pure values parsed from text. There are two other functions, as for signers, called `lookupMaterial` (which returns a `Maybe` rather than throwing an error) and `materials`, which gets a `Map` of all materials with the desired type.

After obtaining the material (the nametag that was entered), we call its `checkNametag` to prove that it is real. This value won't be used until we return.

Then we obtain the identity of the caller and make three definitions:

- `addEntry` is a function that is part of an informal API for the `Entries` type, described below. It returns the number of submissions for the present caller and the new map of entries.
- `count'` and `winningCount'` are both updates to the corresponding fields of `LotteryState`; the logic is self-evident.

Having created the necessary updated values, we construct and insert the new state:

Snippet: updating the state

```
      put $
        if count' == limit
        then let (winners, nonWinners) = splitEntries winningCount' nametags' in
             FinishedState{totalWinners = fromIntegral $ Map.size winners, ..}
        else RunningState
```

```
                {
                  nametags = nametags',
                  count = count',
                  winningCount = winningCount'
                }
```

PUT

Like `get`, `put` is part of the `StateT` API and overwrites the old state with a new one.

The value we choose to overwrite it with depends on where in the lottery we are. If the limit has not yet been reached, then we just replace all the fields of `RunningState` (the one we had when we started `newEntry`) with their primed equivalents. If it has, though, we have to create a new kind of state, which means we need the individual maps of winning entrants and...other...entrants. This is achieved by `splitEntries`, another part of the `Entries` "API".

FROMINTETGRAL

The `totalWinners` value is essentially the size of the `winners` map, but it has to be cast (in C-language parlance) to the correct numeric type, as `Map.size` is an `Int` while `totalWinners` is a `Natural`. Fortunately, an `Int` is an instance of the class `Integral`, which defines `fromIntegral` to convert from it to any other numeric type.

Having update the state, we can now return:

Snippet: returning

```
        return $! EnterResult{..}
```

MORE RECORD WILDCARDS

The record wildcard syntax has a matching assignment syntax that *takes* values named after the fields of, in this case, an `EnterResult`, and assigns them to those fields. Obviously they must be in scope, which is why we carefully chose the names `enterCount` and `message` earlier.

STRICT FUNCTION APPLICATION

The `$!` operator is the *strict application* operator, like `$` but with the additional property that it forces the right-hand side. This, in turn, causes all the `!`-marked fields of `EnterResult` to be forced as well, and this flushes out all the errors that may have arisen in the course of constructing them: a bad nametag (producing an error in `message`) or any of the runtime errors in the code above, all of which must be resolved before `enterCount` can be calculated.

The `Entries` API

Before turning to the `getEntry` function, we stop to describe a small interface for the `Entries` type. Although this is just a combination of several library types, it has a higher-level purpose that must be served by well-named, atomic methods that cut through all the algorithmic noise. In this way, we are effectively defining an object-oriented class, though one whose members are all immutable.

Snippet: `takeEntry`

```
takeEntry :: PublicKey -> Entries -> (Maybe Nametags, Entries)
takeEntry = Map.alterF $ (,Nothing) . fmap Container
```

This removes the entry (a `Seq Nametag`) from the `Map` that is an `Entries`, if it is there, and returns it along with the new `Map`. All the work is done by `Map.alterF`, a function from `Data.Map` that is so high-level that it is difficult even to understand how it works from its type signature. Here is a conversational walk-through of its explanation.

- Each entry in a `Map` may, or may not, be there, depending on whether its key is present. So a `Map a` is a lot of `Maybe a` values.
- A function that alters a `Map` at a single entry, therefore, will employ a function of the form `Maybe a -> Maybe a` that does something with an entry (that may not be there) to produce another one (or delete it). One such function is the one that always returns `Nothing`, meaning that no matter what the entry is, it will be deleted.
- In addition to modifying the `Map`, we also want to return the entry that was there, if it was there, as a `Nametags`, which is a `Container (Seq Nametag)`. If it was there, we are holding a `Just nts`, and we want to return `Just (Container nts)`; if it wasn't, we are holding and want to return a `Nothing`. The idiom for this is `fmap Container`.

The actual function that is passed to `Map.alterF` is a composition, so operates by: first, applying the `Container` to the maybe-entry; and second, adding the boilerplate `Nothing` that instructs `alterF` what to do with the `Map`.

SECTIONS

The odd syntax `(,Nothing)` resembles an incomplete pair, and therefore represents a function that takes what would be the missing first item and filling it in. This is called a *tuple section*, as in, a section (a bit cut out of) a tuple. More generally, there are *operator sections* that allow us to write "half an operator" as a function that fills in the other half: for instance, `(1 +)` is the "add one" function, as is `(+ 1)`. The parentheses are necessary.

FMAP

As mentioned, `fmap` is an operation that takes a function `a -> b` and applies it to the `Just` branch of a `Maybe a`, yielding a `Maybe b`. This applies to other polymorphic types besides `Maybe a`, so long as they are types that, notionally, are at least capable of producing a value of type `a` that can be modified. (These types are technically called `Functor`s, which is the typeclass to which they belong.)

Snippet: `addEntry`

```
addEntry :: Nametag -> PublicKey -> Entries -> (Natural, Entries)
addEntry tag = Map.alterF $ finish . maybe (Seq.singleton tag) (tag Seq.<|)
  where finish set = (fromIntegral $ Seq.length set, Just set)
```

This inserts a new nametag under a possibly-existing identity, returning the number of nametags then present for that identity, as well as the modified `Entries`. It also uses `Map.alterF`, in a rather more complicated way. The altering function, given in the snippet as a composition of two functions, has the overall signature

```
Maybe (Seq Nametag) -> (Natural, Maybe (Seq Nametag))
```

and so as it begins, it considers a `Maybe (Seq Nametag)`, which it handles by supplying an action for each of the two branches (`Nothing` or `Just nts`) and bundling them with `maybe`, described below. This gives a `Seq Nametag` regardless of which branch was originally present (because even if nothing was there, we have now inserted a new nametag). It then applies `finish` to do what the name says: extract the size of that set of nametags (the length of the sequence) and bundle the actual nametags into a `Just` to indicate that we are leaving the `Map` with an updated entry.

MAYBE FUNCTION

The `maybe` function is a self-contained *if*-statement: the *then* branch, the second one, says what to do with `nts` if we get a `Just nts`; and the *else* branch, the first one, says what to do with a `Nothing`. Hence, the *then* branch is a function of

the form `Seq Nametag -> Seq Nametag` that, we see, is another "section" of the operator `Seq.<|`, which, unqualified as `<|`, is intended to represent an arrow indicating which end of the existing sequence `nts` we stick the new nametag onto. Correspondingly, the *else* branch is a single `Seq Nametag` value, which can only be the sequence with one element, the new tag. This construct enforces the requirement implict in a `Maybe` that we must consider both possibilities.

Snippet: `splitEntries`

```
splitEntries :: Natural -> Entries -> (Entries, Entries)
splitEntries n = Map.partition ((== n) . fromIntegral . Seq.length)
```

Here we use a different one of `Map`'s methods, `Map.partition`, which does what it says: given a predicate on entries of the map, creates two values, each containing the entries that do and do not satisfy the predicate. Here, the predicate is "equality with `n` of the length of the `Seq` that is the entry".

Getting entries

In this lottery, entrants are entitled to get their entries back. This can occur either when the lottery is ongoing, in which case they simply withdraw from it, or when it is finished, in which case they withdraw, unless they are a winner, which produces a different response. It is crucial that the contract allow withdrawals while it is ongoing, because the lottery may stall with no new entries before it is finished, and the owner may not be trustworthy, so participants need to be able to "call it" when they decide it's not worth waiting anymore.

Snippet: starting `getEntry`

```
getEntry ::
  Natural -> StateT LotteryState (Fae LotteryAction LotteryResult) LotteryResult
getEntry lotteryLimit = do
  entrant <- signer "self"
  lotteryState <- get
(result, newState) <- case lotteryState of
```

This code just sets up the rest of the function. Note that the return value is exactly the same as that of `newEntry`, as it must be, though in this case, the contract owner's permission is not necessary because no new nametag is going to be validated.

Snippet: withdrawing from a running lottery

```
    RunningState{..} -> do
      let (returnedM, nametags') = takeEntry entrant nametags
          returned = fromMaybe (throw NotFound) returnedM
          result = ExitResult{..}
          count' = count - 1
          winningCount'
            | count < winningCount = winningCount
            | otherwise = fromIntegral $ maximum $ fmap Seq.length nametags'
          newState =
            RunningState
            {
              nametags = nametags',
              count = count',
              winningCount = winningCount'
```

```
            }
      return (result, newState)
```

When the lottery is ongoing, a entrant can withdraw, but it entails a significant amount of recalculation. Everything except `winningCount'` is either self-explanatory or analogous to constructs we have seen before, but that one entails a new library function.

MAXIMUM

The `maximum` function applies to any container-like type (specifically, any instance of the class `Foldable`, meaning, any type whose values can be "summed up", and so in some sense, contain things), and does what the name says: computes the maximum value from among all those in the container, if they are an `Ord`ered type.

We need to use `maximum` to find the new `winningCount` because, unlike in the `addEntry` case, there is no guarantee how much the length of the largest entry will change if an entry with that length is removed. It could not move at all, if there are other winners; or it could drop by any amount if we have removed the unique winner, depending on how many nametags the other entries have, and this amount cannot be determined just from knowing `winningCount`. Therefore, we have to scan the entire `Map`, an operation that we tried to avoid by using `winningCount` in the first place; however, since withdrawing from a running lottery *should* be an uncommon operation, if there is to be inefficiency, this is where to put it. To limit the damage we recognize the "easy" case where the removed entry was not a winner, and don't do the scan there.

Snippet: withdrawing when the lottery is finished

```
    FinishedState{..} -> do
      let (wSetM, winners') = takeEntry entrant winners
          (nwSetM, nonWinners') = takeEntry entrant nonWinners
      case (wSetM, nwSetM) of
        (Nothing, Nothing) -> throw NotFound
        (Just _, Just _) -> throw BothMaps
        (Just returned, _) -> do
          let result = WinResult{winnersCount = totalWinners, ..}
              newState = lotteryState{winners = winners'}
          return (result, newState)
        (_, Just returned) -> do
          let result = ExitResult{..}
              newState = lotteryState{nonWinners = nonWinners'}
          return (result, newState)
```

Here, we try to get the caller's entries from both maps, and depending on what the results of those attempts are, either throw an error (if found in neither or both), or return a `WinResult` or `ExitResult` depending on which single lookup succeeded. All of this is self-explanatory and simply consists of assigning the appropriate fields of the various values.

Snippet: returning

```
  put newState
  return $! result
```

Finally, regardless of what happened, we got a `newState` and a `result`, the former of which we save and the latter of which we (strictly, again) return.

Lottery transactions

The lottery contract being at last complete, we can now provide some transactions that interact with it. Here are the ones that simply invoke the three possible operations: create a lottery, enter a new nametag, or exit.

Transaction source file: NewLottery.hs

```
import Lottery

body :: FaeTX ()
body = do
  owner <- signer "self"
  newContract $ Lottery 5 owner
```

Transaction message: NewLottery

```
body = NewLottery
others
  - Nametag
  - Lottery
keys
  self = $self
```

The transaction `body` module imports our `Lottery` module, which contains all the snippets above (as well as the necessary module imports). It sets up a lottery for 5 entries, owned by whomever signed the transaction, which is determined by the `$self` environment variable in the message file. Observe that the message file must include the `Lottery` module as an attachment; we also upload `Nametag` for this example, in case it is not present already. In "reality", there should be an origin transaction for the `Nametag` type that installed this module and to which all other transactions using it can refer.

Transaction source file: EnterLottery.hs

```
import Blockchain.Fae.Transactions.TX$lotteryID.Lottery

body :: LotteryResult -> FaeTX (Natural, String)
body EnterResult{..} = return (enterCount, message)
```

Transaction message: EnterLottery

```
body = EnterLottery
keys
  self = $self
inputs
  $lotteryID/Body/0/Current : Enter
    self = self
    nametag = $nametagID/Body/0/Current : ()
      self = self
```

This transaction is easier to read with the message first. It, too, includes a `self` signer, who of course is probably different from the one who signed `NewLottery`. It has a single input call, to the contract that we know is the index-0 output of the `body` of `NewLottery`, which has some transaction ID `$lotteryID`. This call gets `Enter` as an argument, naturally. It also declares a signer name remapping `self = self` and a materials call assigning `nametag`, both described below.

SIGNER REMAPPINGS

Input contracts do not inherit the `signers` map of their enclosing transaction, but have to declare which new names (on the left) need to be assigned from old ones or from literal public keys (both on the right). In this case, we want to simply pass along `self` under the same name. This feature is also present in the `useEscrow` function inside contract or transaction code: the first argument, which until now has been an empty list `[]`, actually contains a number of remappings, each of the form:

```
newSignerRole <-| oldSignerRole  or  newSignerRole ↤ oldSignerRole (↤ = U+21a4)
```

This allows us to pass along signers that will be present in the escrow call. It is this situation that accounts for the restriction that only explicitly remapped names are present in the call: otherwise, it would be possible for a back-door signer or material to alter the behavior of some deeply nested call without the possibility of any intermediate calls preventing this.

MATERIALS CALLS

Materials, which we saw could be obtained in a contract using `material`, are actually supplied via *named contract calls* alongside the signer remappings (they can also appear in a top-level section `materials`, with the same syntax, and then they appear as materials in the transaction body). Here, we call a contract produced in transaction `$nametagID` that, apparently, caused a `Nametag` to be stored, such as via `deposit`. This value will appear when\`material "nametag"` is called in the lottery contract.

Materials come with two more remapping syntaxes for `useEscrow`:

```
newMaterialName <=| oldMaterialName  or  newMaterialName ⇤ oldMaterialName (⇤ = U+290
newMaterialName *<- materialValue    or  newMaterialName ↞ materialValue   (↞ = U+291
```

The first one is the same as for `signers`; the second one is present because it is possible that the escrow expects a material to be present and the desired value in the calling code was obtained, not as a material, but inline somehow.

HOW IT WORKS

Having understood the transaction message, the `body` function simply accepts the result of calling the lottery contract and prints (returns) the corresponding fields. Note that it does not handle either of the `Exit`-associated results, so technically this `body` is a partial function, but we are guaranteed by the contract's author that it works this way, so we do not expect a pattern mismatch. Note the use of `import` to get the `Lottery` module from the `NewLottery` transaction.

Transaction source file: ExitLottery.hs

```
import Blockchain.Fae.Transactions.TX$lotteryID.Lottery

body :: LotteryResult -> FaeTX String
body WinResult{..} = return $ "Won with " ++ show others ++ " other" ++ plural where
  others = winnersCount - 1
  plural | others == 1 = ""
         | otherwise = "s"
body ExitResult{..} = return $ "Did not win out of " ++ show lotteryLimit ++ " entries
```

Transaction message: ExitLottery

```
body = ExitLottery
keys
  self = $self
inputs
  $lotteryID/Body/0/Current : Exit
    self = self
```

The transaction message is simpler than for `EnterLottery`, because we do not supply any materials, and all the other aspects of it are the same except for the argument to the lottery contract, which is now of course `Exit`.

The `body` has two cases, one to handle each kind of exit response, and produces a nicely (in the case of `WinResult` almost obsessively nicely) formatted message describing what happened.