Transactions (in databases) are sequneces of read/write operation, which fulfill the ACID property:

- A tomicity
- C onsistency
- I solation
- D urability

Software transactional memory (STM), uses this idea for communication between concurrent processes/threads. Only durability is not guaranteed, since it is a concept for working on data in memory.

Use TVar instead of an MVar. A TVar is simily to a cell in a data base and cannot be empty. Operations to read and write TVars with the STM monad.

An STM action can be started within the IO monad with the function
    atomically :: STM a -> IO a

If a transaction reaches an unexpected point, it can be skiped be calling retry :: STM ().

Transactions can be composed as

- as sequences
- in combination with branching
- by `orElse`, which adds an alternative transaction, only executed if the first transaction fails in retry.

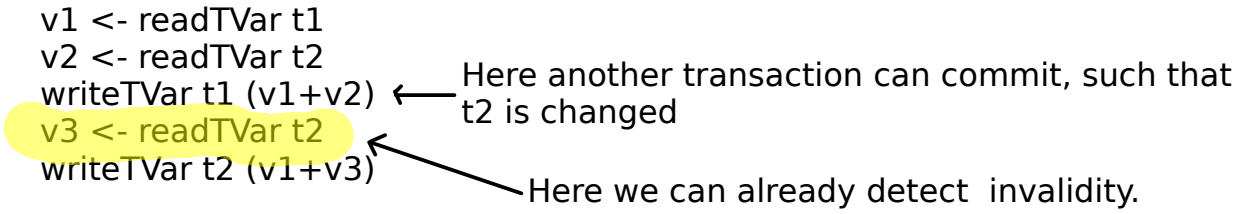
How can we implement STM?

In data bases you usually use pessimistic locking. This is not so appropriate for software transactions. We want to execute processes as much in parallel as possible.

Instead we use an optimistic strategie, which means run as much in parallel as possible and restart in case of an occurring problem.

Collect all writeTVar information to the end of the transaction. There we can realize it in one step (locking needed here), which is called the commit phase.

A transaction can be invalid in the end, so we may not commit.



So before committing, we have the check, whether all read TVars are still valid. Therefore, we also collect the read values during the execution and check, whether they are still valid, before we commit.

Collect ReadSet and WriteSet during the execution. Use the ReadSet for the validity check and the WritSet for the commit. If a transaction is invalid, we have a rollback, which directly restarts the transaction.

Locking in the commit phase can avoid deadlocks with the two techniques known from the philosophers. We know all TVars relevant for the transaction and can lock them (both from ReadSet and WriteSet) either with respect to a global order or by putting back all locks taken so fare, if the next one isn't available, and then directly try again.

Consider the follwoing case:

```
v1 <- readTVar t1
writeTVar t1 (v1+1)
v2 <- readTVar t1
```