Assignment 2

Alessandro Puccia 547462

To model the security policies I used the DFAs implementation and the helper functions provided with the assignment. The alphabet sigma that represents the security-relevant operations is fixed and composed of 4 operations that are used in the examples: `w` for Write, `r` for Read, `o` for Open and `e` for Execute.

In addition to the helper functions provided with the DFA implementation, I defined another function called `checkPolicies` that takes in input the history of execution and a list of DFAs to check that each policy is satisfied.

Syntactic constructs added

Given the base interpreter of the previous examples, I added the following constructs:

- `Phi` takes the states of the DFA, the initial state, the transitions and the accepting states. This construct is used to allow the users to define new policies, the alphabet `sigma` is predefined.
- `Frame` takes a security policy and an expression in which the policy must be satisfied.
- Some constructs that emulates security-relevant operations: `Open`, `Write`, `Read` and `Execute`.

Value constructs added

A construct `PhiVal` is added that takes the `dfa` type. This construct is used, for example, to save in the environment the policies and to use them later.

Changes to the `eval` interpreter

The interpreter now takes in input two additional parameters:

- `history`: a `string` value that is used to represent, using the predefined sigma symbols, the history of execution about the operations that are relevant for security.
- `activePhis`: a `list` value that will contain all the policies that are enabled in a given moment using the `Frame` construct.

The interpreter also returns a tuple `value` * `string`, the second component is the updated history after an operation.

For each of the constructs that mimic the security-relevant operations first, it is checked that the updated history satisfies all the active policies, using the `checkPolicies` function, and then execute a fake operation.

The construct used to create the policies simply create a `dfa` value using the data provided by the user, then wraps it into a `PhiVal` construct.

Inside the `Frame` construct, first I check that the policy that is passed is effectively evaluated as a policy wrapped inside the `PhiVal` construct and then if the history is empty simply continue the evaluation (appending the policy in the list `activePhis`). If the history is not empty then it is checked using the provided function `checkAccepts` if the past history does not satisfy the policy the execution is stopped and the

following `expr` is not evaluated, otherwise the evaluation continues (appending the policy in the list `activePhis`).