KRR Project 1

due January 6th, 2013

For this project you are going to implement an application for working with modal formulas and Kripke models. The application must have at least a command line interface able to recognize and execute the commands presented in this file. While the actual execution of individual commands may be replaced with graphical-based interaction (menus, buttons, etc.), it is very important that your application can also execute commands from a file:

```
EXECUTE COMMANDS FROM file_name
```

This command should read the input file and execute all the commands within it. Each command will be given on a separate line and will obey the same syntax as that indicated for use in the command line. If your application cannot execute commands from a file, you will be penalized with 25% of the score for this project.

This project is worth 20p (out of 100p) from your final grade for the KRR course. The project is split into three parts, worth 8p, 6p and 6p, respectively. You can also get up to 5p bonus (see the corresponding section for details).

1 Kripke models – 8p

The first part of the project refers to manipulating Kripke models. Recall that a Kripke model consists of the following elements:

- a set of worlds W
- a binary accessibility relation $R \subseteq W \times W$
- a valuation function $V : \mathcal{P}rop \to 2^W$ that gives, for each proposition $p \in \mathcal{P}rop$, the set of worlds $V(p) \subseteq W$ where p is true.

Note that the valuation function V needs the set of all propositions available in the language. Since we are going to build the models step by step, this set might not be available from the beginning, so we are going to rely on an equivalent representation. We will use the function $True : W \to 2^{\mathcal{P}rop}$ to return, for each world $w \in W$ the set $True(w) \subseteq \mathcal{P}rop$ of propositions that are true at w.

The rest of this section lists all the commands related to Kripke models that should be recognized by your application. All commands will be written using capital letters. The names for models, worlds and propositions will only use lowercase letters, digits and the underscore character.

1.1 Create, edit, display – 1p

```
CREATE MODEL model_name
```

- creates the model `model_name`
- if the model already exists, prints the message "Model `model_name` already exists. Use the EDIT command to open this model for editing."

```
CREATE MODEL model_name FORCE
```

- creates the model `model_name`
- if the model already exists, prints the message "Existing model `model_name` overwritten." and removes all the previous content of the model

`REMOVE MODEL model_name`

- removes the model `model_name` from memory
- if the model does not exist, prints the message "Model `model_name` not found."

`LIST MODELS`

- prints "Models : " followed by a space-separated list containing the names of all models that are currently in memory

`SHOW MODEL model_name`

- displays the content of model `model_name`
- if the model does not exist, displays the message "Model `model_name` not found."
- an empty model is displayed as follows:

```
Model m1 :
  Worlds = {}
  Relation = {}
```

- a non-empty model is displayed as follows:

```
Model m2:
  Worlds = {w1, w2, w3}
  Relation = {(w1, w2), (w2, w3)}
  True(w1) = {p, q}
  True(w2) = {}
  True(w3) = {r}
```

`SHOW ALL MODELS`

- displays the content of all models, using the convention suggested above

`EDIT MODEL model_name`

- activates model `model_name` for editing; all the editing commands will alter the content of this model until the next `EDIT` command
- if the model does not exist, displays the message "Model `model_name` not found."

`SHOW`

- displays the content of the model that is currently open for editing
- if no model is open for editing, displays the message "No model is currently open for editing. Use the EDIT command first." **All the editing commands from the rest of this section display the same message if no model is currently open for editing.**

1.2 Create and edit worlds – 2p

`ADD WORLDS world_1 world_2 ... world_n`

- adds the given worlds to the active model
- for each world `world_i` that already exists in the model, displays the message "World `world_i` already exists."

`REMOVE WORLDS world_1 world_2 ... world_n`

- removes the given worlds from the model

- also removes all accessibility relation pairs that contain one of the given worlds
- for any world `world_i` that was not part of the model, displays the message "World `world_i` not found."

```
RENAME WORLD old_name TO new_name
```

- renames all occurrences (in the list of worlds, the accessibility relation or the truth assignment) of the world `old_name` to `new_name`; does not check for name clashes (the world `new_name` might already exist in the model – remove all duplicate entries in this case and take the truth set of `new_name` to be the union of what it was before and the truth set of `old_name`)
- if the world `old_name` was not part of the model, displays the message "World `old_name` not found."

```
CLEAR
```

- removes all the content from the active model (worlds, accessibility relation and truth assignment)

1.3 Create and edit accessibility – 1.5p

```
ADD RELATIONS world_1_a world_1_b ... world_n_a world_n_b
```

- adds the pairs `(world_i_a, world_i_b)` to the accessibility relation of the active model
- for each pair `(world_i_a, world_i_b)` that was already in the model, displays the message "Pair `(world_i_a, world_i_b)` already exists."
- for each world `world_i_a` (or `world_i_b`) that does not exist in the model, displays the message "World `world_i_a` not found."

```
REMOVE RELATIONS world_1_a world_1_b ... world_n_a world_n_b
```

- removes the pairs `(world_i_a, world_i_b)` from the accessibility relation of the active model
- for each pair `(world_i_a, world_i_b)` that was not in the model, displays the message "Pair `(world_i_a, world_i_b)` not found." (this includes the case when one or both worlds from the pair are not part of the model)

```
CLEAR RELATIONS
```

- removes all the pairs from the accessibility relation for the current model
- the set of worlds and the truth values are not affected

1.4 Create and edit truth value – 1.5p

```
ADD TRUE prop_1 ... prop_k AT world_1 ... world_n
```

- for each `prop_i` and `world_j`, adds `prop_i` to `True(world_j)`
- if `prop_i` is already in `True(world_j)`, prints the message "Proposition `prop_j` already true at world `world_j`."
- for each world `world_i` that does not exist in the model, prints the message "World `world_i` not found."

```
REMOVE TRUE prop_1 ... prop_k AT world_1 ... world_n
```

- for each `prop_i` and `world_j`, removes `prop_i` from `True(world_j)`
- if `prop_i` was not in `True(world_j)`, prints the message "Proposition `prop_j` was not true at world `world_j`."
- for each world `world_i` that does not exist in the model, prints the message "World `world_i` not found."

```
REMOVE PROPOSITIONS prop_1 prop_2 ... prop_n
```

- removes each proposition `prop_i` from the truth set of every world of the model

`RENAME PROPOSITION old_name TO new_name`

- renames all occurrences of proposition `old_name` in the truth sets to `new_name`; does not check for name clashes (the proposition `old_name` might already be part of the model)

`CLEAR TRUTH`

- clears all truth assignment information from the active model (all truth sets will be emtpy)

1.5 Load and save – 2p

`LOAD MODELS FROM file_name`

- loads Kripke models from a file (models with the same name are overwritten)
- if the file is not found, prints the message "File '`file_name`' not found."
- the format of the input file is the following:
  - the first line contains the number of models; the following lines will give the content of exactly that many models (assume that the input file is correct)
  - the first line from a model definition gives the model name, preceded by the keyword `MODEL`
  - the second line gives a space separated list of the worlds of the model, preceded by the keyword `WORLDS`
  - the third line gives the accessibility relation of the model, as a list of space-separated pairs, preceded by the keyword `RELATION`; each pair is given as two worlds separated by a comma and enclosed in parentheses
  - the truth assignment is given on separate lines for each world from the model, by having the keyword `TRUE` followed by the world name and a colon, then the list of propositions that are true in the given world, separated by spaces
- as an example, see the following:

```
2
MODEL m0
WORLDS
RELATION
MODEL m1
WORLDS w1 w2 w3
RELATION (w1, w2) (w2, w3)
TRUE w1 : p q
TRUE w2 :
TRUE w3 : r
```

`SAVE MODELS TO file_name`

- save all models to a file, using the same format as above
- overwrite existing files

2 Modal formulas – 6p

The second part of the project requires that you properly read and evaluate modal formulas. The simplest modal formulas are `T` (the true constant), `F` (the false constant) and `p`, where `p` is a proposition. Proposition names may contain any combination of lowercase letters, digits and the underscore character. If `$phi` and `$psi` stand for well-formed modal formulas, then the following are also well-formed modal formulas:

```
($phi)
~$phi
$phi & $psi
$phi | $psi
$phi -> $psi
$phi <-> $psi
<>$phi
[]$phi
```

2.1 Create and display – 3p

`CREATE FORMULA f1 = expression`

- creates the formula named `f1` and gives it the value encoded by `expression`
- overwrites existing formulas
- example

```
CREATE FORMULA f1 = p
CREATE FORMULA f2 = p & ~q
CREATE FORMULA f3 = <>(p & [](q | ~r))
```

- improved behavior (worth 1.5p of the 3p): the expression may contain previously defined formulas, written with their name preceded by `$`
- example

```
CREATE FORMULA f4 = $f1 & $f2
```

- for any reference `$name` to undefined formulas, prints the message "Formula `$name` not found." (does not create any formula in this case)

`SHOW FORMULA name`

- prints the formula `name` using the same syntax conventions as those used for creating formulas
- if the formula does not exist, prints the message "Formula not found."
- for example, the output for the formula `f4` defined above is:

```
p & p & ~q
```

2.2 Evaluate – 3p

`EVALUATE $formula_name IN model AT world`

- evaluates formula `formula_name` in the model `model` at world `world` and outputs either `TRUE` or `FALSE`
- if the formula does not exist, prints "Formula `formula_name` not found."
- if the model does not exist, prints "Model `model` not found."
- if the world does not exist in the model, prints "World `world` not found in model `model`."

`EVALUATE $formula_name IN model`

- evaluates a formula at all worlds of the given model
- outputs a comma separated list of worlds where the formula is true, enclosed in braces, such as `{w1, w2}` or `{}` (list may also be empty)
- prints the same error messages as above if the formula or the model do not exist

In order to receive full points for this part, the evaluation of formulas must be correct. The precedence of the operators is the following:

```
[], <>, ~   - highest precedence
&
|
->
<->         - lowest precedence
```

The evaluation of binary operators of the same precedence is done from left to right (the rightmost one is the last to be computed). You should also account for parentheses in the formulas.

3 Justified evaluation – 6p

In the third part of the project you are required to provide a justification for the truth value of a formula. The following command requires a one-step justification:

```
EVALUATE $formula_name IN model AT world WHY 1
```

The one-step refers to the fact that only the largest subformulas of a formula are considered for the justification. The rules for one-step justification follow.

The evaluation of a constant does not need any justification, so the output will just provide the corresponding truth value, as follows

```
T is TRUE at w_0
F is FALSE at w_0
```

The evaluation of a proposition depends on whether the proposition is true or not in the given world, based on the truth assignment from the model. This leads to the following possible outputs:

```
prop is TRUE at  w_0
    * prop is in True(w_0)
prop is FALSE at w_0
    * prop is not in True(w_0)
```

The negation of a formula is true if and only if the formula is false, and the other way around. This leads to the following possible justifications:

```
~p is TRUE at w_0
    * p is FALSE at w_0
~p is FALSE at w_0
    * p is TRUE at w_0
```

The conjunction of two formulas is true iff both of them are true. The conjunction is false iff one of them is false (if the first one is false, that justification should be used).

```
p & q is TRUE at w_0
    * p is TRUE at w_0
    * q is TRUE at w_0
p & q is FALSE at w_0
    * p is FALSE at w_0
p & q is FALSE at w_0
    * q is FALSE at w_0
```

A disjunction of two formulas is true iff any of them is true (if the first one is true, the corresponding justification should be used). The disjunction is false iff both formulas are false.

```
p | q is TRUE at w_0
    * p is TRUE at w_0
p | q is TRUE at w_0
    * q is TRUE at w_0
```

```
p | q is FALSE at w_0
    * p is FALSE at w_0
    * q is FALSE at w_0
```

The implication is true iff the first formula is false or the second one is true (pick the first justification that holds). The implication is false iff the first formula is true and the second one is false.

```
p -> q is TRUE at w_0
    * p is FALSE at w_0
p -> q is TRUE at w_0
    * q is TRUE at w_0
p -> q is FALSE at w_0
    * p is TRUE at w_0
    * q is FALSE at w_0
```

Similarly to the above, there are two possible justifications for each truth value of the equivalence of two formulas.

```
p <-> q is TRUE at w_0
    * p is TRUE at w_0
    * q is TRUE at w_0
p <-> q is TRUE at w_0
    * p is FALSE at w_0
    * q is FALSE at w_0
p <-> q is FALSE at w_0
    * p is TRUE at w_0
    * q is FALSE at w_0
p <-> q is FALSE at w_0
    * p is FALSE at w_0
    * q is TRUE at w_0
```

The formula `<>p` is true at a world `w_0` iff there exists a world `w_i`, accessible from `w_0`, such that `p` holds at `w_i`. The formula `<>p` is false iff `p` is false at all worlds `w_ij` accessible from `w_0`.

```
<>p is TRUE at w0
    * w_i in next(w_0)
    * p is TRUE at w_i
<>p is FALSE at w0
    * next(w_0) = {w_i1, ..., w_ik}
    * p is FALSE at w_i1
    * ...
    * p is FALSE at w_ik
```

The formula `[]p` is true at a world `w_0` iff `p` is true at all worlds `w_ij` accessible from `w_0`. The formula `[]p` is false iff there exists a world `w_i` such that `p` is false at `w_i`.

```
[]p is TRUE at w0
    * next(w_0) = {w_i1, ..., w_ik}
    * p is TRUE at w_i1
    * ...
    * p is TRUE at w_ik
[]p is FALSE at w0
    * w_i in next(w_0)
    * p is FALSE at w_i
```

A two-step justification consists of the truth value of the formula, a one-step justification for the truth value and also one-step justification for each justification. This can also be extended to n-step justification.

```
EVALUATE $formula_name IN model AT world WHY n
```

- prints the n-step justification of the formula

```
EVALUATE $formula_name IN model AT world WHY ALL
```

- print the complete justification of the formula, i.e. recursively justify all justifications

For example, consider the following model (as it would appear in an input file):

```
MODEL m1
WORLDS w1 w2 w3
RELATION (w1, w2) (w2, w3)
TRUE w1 :
TRUE w2 : p
TRUE w3 : p
```

Also, consider the formula:

```
f = !p & []p & <><>p
```

Then the two-step justification of the `f` is

```
!p & []p & <><>p is TRUE at w0
    * !p & []p is TRUE at w0
        * !p is TRUE at w0
        * []p is TRUE at w0
    * <><>p is TRUE at w0
        * w1 in next(w0)
        * <>p is TRUE at w1
```

The full justification of `f` is:

```
!p & []p & <><>p is TRUE at w0
    * !p & []p is TRUE at w0
        * !p is TRUE at w0
            * p is FALSE at w0
                * p not in True(w0)
        * []p is TRUE at w0
            * next(w0) = {w1}
            * p is TRUE at w1
                * p in True(w1)
    * <><>p is TRUE at w0
        * w1 in next(w0)
        * <>p is TRUE at w1
            * w2 in next(w1)
            * p is TRUE at w2
                * p iN True(w2)
```

4 Bonus – up to 5p

You can receive up to 5 bonus points for additional features that you implement in your application. Improvements may include, but are not limited to:

- graphical representation of the Kripke models
- interaction with the graphical representation of a Kripke model for editing part of it
- graphical representation for the justifications (tree or otherwise)
- additional manipulation instructions for models or formulas (normal form of a formula, evaluation of a formula for all possible truth assignments of the same model, etc.)

5 Final remarks

You may use any programming language. There is also a test suite for this project, consisting of several text files. Running the commands from "commands.txt" (which will also load the models from "input.txt") should generate the file "output.txt" and produce a console output similar to that provided in "console.txt". While the results need not be exactly the same (up to number of spaces or so), outputs that are significantly different will be penalized.