AMPL syntax reference

File types:

- $\boxed{\texttt{.mod}}$ file: file with the description of a mathematical *problem*
- $\boxed{\texttt{.dat}}$ file: file with numerical data describing a specific *instance* of the problem
- $\boxed{\texttt{.run}}$ file: file with a sequence of AMPL commands

Model file

Contains definition of *indices*, *parameters*, *decision variables*, *objection functions*, *constraints.*

1. **Index sets** contain indices where variables and parameters are defined. Example: if x is in $\mathbb{R}^n$, the set I will contain the indices i for which x_i is defined (e.g., $1, \ldots, n$). The content of I *is not specified* in the model file.

   Keyword: $\boxed{\texttt{set}}$.

   Syntax:
   `set I;`

2. **Parameters**: everything that is part of the problem but is not a decision variable.

   Keyword: $\boxed{\texttt{param}}$.

   If the parameter is a scalar:
   `param myScalar.`

   If it is a vector or a matrix, you need to specify the set(s) of indices where it is defined:
   `param myVector{I};` or `param myVector{i in I};`
   `param myMatrix{I,J};`

   In the second case, `myMatrix` is defined for each index in the cartesian product $I \times J$.

   NOTE: The value of the parameters should not be specified in the model file, but it is sometimes useful to specify a *default* value for the parameters (i.e., a value which will be used unless a different value is specified in the `.dat` file). This is done with the keyword $\boxed{\texttt{default}}$. Example: if you know that matrix A contains all entries of value 1 (unless a subset of elements, which are specified in the `.dat` file), you can write:
   `param A{I,J} default 1;`

3. **Decision variables**.

   Keyword: $\boxed{\texttt{var}}$.
   The indices where the variable is defined are specified as for the parameters.

   NOTE: Decision variables are NOT "programming" variables, and you *cannot* assign values to decision variables. Variables in AMPL are symbols used in defining the model, that will contain a value only after the problem has been solved.

   Syntax:
   `var myVar{I};`

   Positivity, negativity, box constraints (lower and upper bound), integrality, binarity are imposed as follows:

```
var myVar{I} >= 0;
var myVar{I} >= 5, <= 18;
var myVar{i in I} >= lb[i]; *
var myVar{I,J} integer;
var myVar{I,J} binary;
var myVar{I,J} >= 0, integer;
```

4. **Objective function**.

   Keyword: `minimize` or `maximize`.

   Example:
   `minimize myFunction:  sum{i in I} myVar[i]*myVector[i];`

   NOTE: The symbol ":" must be put before the definition of constraints/objective function.

   NOTE: Each objective function and constraints must have a unique name.

   NOTE: It is good practice to usa capital letters for the sets, e.g. `I`, and lower case for the indices, e.g. `i`. Of course, nothing prevents you from using, e.g., `j in I` or `i in J`.

5. **Constraints**.

   Keyword: `subject to` or `s.t.`.

   If we want to define a family of constraints, identified by a set of indices, that set is specified with { } after the constraint name. The index must be specified with the usual syntax `j in J`, and the index `j` can be used in the constraint expression. Example:

   ```
   subject to myConstraint{j in J}: sum{i in I}myMatrix[i,j]*x[i] >= 0;
   ```

   will generate a constraint of the form `myConstraint` for each element $j \in J$.

   It is possible to define a family of constraints only on a subset of indices that satisfy a given condition (which does not depend from the value of the variables). This can be done simply by adding conditions after the declaration of the indices, as follows:

   ```
   s.t. filteredConstraint{i in I, j in I: i<>j and c[i,j]==2}:
         x[i] <= x[j];
   ```

   that is: constraint `filteredConstraint` is defined for each (i, j) such that $i \neq j$ and $c_{ij} = 2$.

   Note that when you filter the indices in this way, you can use logical operators, since you are filtering *a priori*, based on the values of the data (not of the variables!). Of course, you cannot use logical operators in the constraints.

Data file

1. **Index sets**. Example:
   `set I := January February March;`

[*] Explicitly using an index `i` is, in general, not necessary, but useful in case the bounds are different for each i

```
set J := Alice Bob;
set I := 1 ..  10;
```

NOTE: In a data file, it is not necessary to use double quotes `""` around strings. The white spaces are considered the delimiteres between elements of the sets.

2. **Scalar parameters**:
   `param myScalar := 100;`

3. **Vectors**:

```
param    myVector1 :=
January    12
February   31
March      51;
```

NOTE: `January`, `February`, `March` are *indices*!

4. Vector parameters, defining multiple vectors at the same time (sometimes useful):

```
param:    myVector1 myVector2 :=
January     12        0.1
February    31        0.15
March       51        0.98;
```

NOTE: Observe the presence of “`:`”. `January`, `February`, `March` are *indices*, while `myVector1` and `myVector2` are names of the vectors.

5. Matrices:

```
param myMatrix:     Alice    Bob :=
January             0.12     0.31
February            0.51     0.41
March               0.61     0.98;
```

NOTE: observe the different position of “`:`” compared to the previous case. Here `Alice` and `Bob` are indices belonging to the set `J`.

Calling AMPL and solving a problem

To solve the problem defined by an AMPL model, there are two options:

1. Open AMPL shell by calling `ampl`. Once the shell is running, you can write commands to load the model and the data, solve the problem and display the variables

2. Use a `.run` file with a sequence of commands to be executed and call `ampl FILE.run` (On windows, with the provided version of Gusek you can just click on “Go” and it will call `ampl FILE.run`, where FILE.run is the file you are visualizing.)

Example of an AMPL shell session:

```
$ ampl
$ ampl: model myModel.mod;
$ ampl: data myModel.dat;
$ ampl: option solver cplex;
$ ampl: solve;
$ ampl: display x;
$ ampl: quit;
```

NOTE: a command is not ended until a `";"` is encountered!

The commands `model` and `data` load the model and data file. The string `option solver cplex;` tells AMPL it should use the solver CPLEX. The command `display` outputs the value of the variable `x` in the solution found by the solver. To display multiple variables, one can use: `display x, y;`

Notice that the operating system must *know* where `cplex` (or any other solvers) is. The same when you call `ampl` in your terminal. Solutions are:

1. Use the complete path. Example:
   - call ampl with `/full/path/FOR_lab/ampl` in the terminal
   (or `"./ampl"` if it is in the current directory)
   - use `option solver "/full/path/FOR_lab/ampl";` in the AMPL shell or in your `.run` file.
   (Within AMPL commands, always use double quotes `""` around long expressions, especially if the string contains non-alphanumeric characters.)

2. Set the `PATH` system variable to include also the path where ampl and cplex are. On unix systems, this is done with something like:
   `export PATH=$PATH:/full/path/FOR_lab`
   If you want to set the PATH variable once and for all, you can add the line to the file `.profile` or `.bash_profile` in your home folder (create one if it doesn't exist).

A sequence of commands, as in the example above, can be written in a `.run` file that can be called from the terminal, via `ampl`, as follows:
`ampl myProblem.run`
If you want to run a `.run` file from inside the AMPL shell, use:
`ampl: include myProblem.run`

This is sometimes handy, because it lets you run a sequence of commands and, then, it allows you to display/manipulate (in a basic way) the results interactively. Example:

```
$ ampl
$ ampl: include myProblem.run;
    ... loads model and data, run optimization ...
$ ampl: display x, y;
$ ampl: display constraint.slack;
$ ampl: display myVar["January"];
$ ampl: display 12*myVar["January"] + 25;
$ ampl: quit;
```

Note that, in the AMPL shell, you must use use double quotes `""` around strings defining indices.