Tips for using AIMMS

The AIMMS help file is relatively difficult. You have to know what you are looking for. Therefore we give some tricks that are useful within AIMMS. The following things can be used both in the definition section of the declaration blocks and inside procedures. See for example the procedures in the *Cutting Stock*-example to see how sets can be manipulated.

set To formulate a model first one has to define the GROUND SETS. What are we working on? As these are often index sets we may want to use simply numbers. However, even a set that seems to contain simply numbers, could actually be treated by AIMMS as a set of strings. If elements of a set are really integer numbers one should declare that the set is a `subset` of the `Integers`.
A set of integers can be defined in the definition section by for instance

```
data { 1, 2, 3, 4, 5, 6, 7, 8 }
```
or
```
{ 1 .. 8 }
```
or by `ElementRange(1,8)`
or equivalently:
```
ElementRange(from: 1, to: 8)
```

Note that actually, the definition

```
ElementRange(from: 1, to: 12)
```

will create the set $\{01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12\}$. Often it is better to distinguish between different kinds of elements by using a proper prefix. For instance, suppose you work with eight customers, we would like to work with a set
`{C-1,C-2,C-3,C-4,C-5,C-6,C-7,C-8}`, which is easily defined by

```
ElementRange(from: 1, to: 8, prefix: 'C-')
```

If the number of customers is a parameter `NumberOfCust`, the customer set can be defined by

```
ElementRange(from: 1, to: NumberOfCust, prefix: 'C-')
```

When a set is filled in a procedure use the assignment `:=` as in

```
Customers := ElementRange(from: 1, to: NumberOfCust, prefix: 'C-') ;
```

index An item in a set is referred to by use of the index. Indicating that in the set S an element is indexed by i, each appearance of i will automatically treat i as an element of S. Whenever appropriate, an action like `sum[ i, ..]` or `max[i, .. ]` will be carried out over all values $i \in S$. Similar, a statement like `x(i) := 1` is executed for **all** i in S. So here is no need to use an explicit for–loop.
It is sometimes convenient to have more than one index automatically pointing to S. Click the index wizard, and click `new` to add additional indices, for instance $i1$ or $i2$ or ii. Note that an index is fixed to a set, so you cannot reuse an index i for different sets. The index referring to `John` in the set `boys := { Pete, John, Harry }` can be used as follows:

```
P('John') := 20;  P(i | i<>'John') := 10;
```

Note that an index behaves like a pointer in C. This means for instance that the value `P('John'+1)` refers to and is equal to `P('Harry')`. Note that `P('John'+2)` refers to an element beyond the set boundary so it is outside the range and will be skipped. It is possible to work through the set in a cyclic way: the value `P('John'++2)` refers to and is equal to `P('Pete')`.

restriction If an action is to be limited to only a subset of the elements, we can use the *restriction* symbol `|`
For instance we want the add values $Q(i)$ over i in S with the property that value $P(i)$ is at least 5. Then we can use

```
sum[ i | P(i)>=5, Q(i) ]
```

Restrictions can also be applied to formulate constraints or parameters for a limited set of elements. For instance it makes sense to have variables $X(i, j)$ for each pair of elements i and j, with i and j different by defining variable $X(i, j)$ and change in the menu the indexing pair `(i,j)` by

```
(i,j) | i<>j
```

ord Whenever a set S has been defined, with index i, and the set is defined by filling it with objects, for instance $S = \{\text{Amsterdam}, \text{Rotterdam}, \text{Utrecht}\}$, then (internally the set is ordered and) each of these three elements has a position or rank in this order. The rank of element i is given by

```
ord(i)
```

The following assigns to an integer `MinPos` the position of the first city in S for which $P(i) > 3$:

```
MinPos := min[ i | P(i)>3, ord(i) ]
```

If you would like to have the first element i in S for which $P(i) > 3$, the assignment needs an object of type `element parameter`, say with the name `FirstCity` and would be used as follows:

```
FirstCity := argmin[ i | P(i)>3, ord(i) ]
```

order by It may be that you want to work with a set where the elements have a prescribed order. For instance, we have the set of student names and idnumbers participating in the class, and they were collected by creating a set of studentnames, indexed with `stname`, say, together with a string parameter `stid(stname)`. From this we could define the set of idnumbers by filling in the definition block: `{ stid(stname) }`. In the line above we would then add `val(stid)`. The function `val()` tries to associate with the string or element argument a numeric value. Therefore it will treat the student-ids as integer numbers and sort them in ascending order accordingly.

set actions Within a set the first and last element can easily be addressed by `First` and `Last` respectively, so

`First(boys)` and `Last(boys)` refer to
`Pete` and `Harry`, respectively

The size of a set is given by its *cardinality* so `Card(boys)` has value `3`.
A set can be extended in a procedure by

```
boys += { Cor }
```

The scope of an index can be restricted by use of `in` and/or the *set-minus* `-` as in

```
for b in (boys-{Harry}) do ... ... ... endfor;
```

composite sets There are cases in which we would want to work with a set that is (contained in) the Cartesian product of two or more sets: say `BlackFields` should represent the black fields on a chess board. Then we may define first the sets `RowLabels` by filling in its definition block `elementRange(1,8)`, and the set `ColLabels` by filling in its definition block: `data {A,B,C,D,E,F,G,H}`, with indices `rw` and `cl` respectively. Now the set `BlackFields` is first defined to be a subset of `ColLabels x RowLabels`. This is done by selecting the wizard along `Subset of`, within the new window again selecting the wizard; then from the menu of already defined sets select `ColLabels`. Now use the arrow-down button to move `ColLabels` to the Compound-Set-box below. Use the wizard once more to select the set `RowLabels`, and move it to the Compound-Set-box by the arrow-down button. Notice that in this way you can create an arbitrary big Cartesian product of a mixture of sets. Use the up- and down arrows to move the constituents of the product in the right order. If ready press OK to save the Cartesian product. Now let us assume we use `bf` as index. Now we fill the definition block with the expression `{ (cl,rw) | Mod(ord(cl),2)=Mod(ord(rw),2) }`. This way we select exactly the black fields of the chess board.

Now check-commit-close the declaration of the set BlackFields and after that *re-open* it! You will notice that a field has been added to the menu, named *Tags*, just below *Subset of*. Here we can choose to fill in `(let,dig)`. These refer to the two coordinates each element of `BlackFields` has. Here we call them after *letter* and *digit* as this is the standard labeling of chess board fields. In any remaining part of the model an element `bf` has a first coordinate `bf.let` (which is an element of the set `ColLabels`) and a second coordinate that can be referred to as `bf.dig`.