AIMMS Language Reference - Set Declaration

AIMMS 4

AIMMS B.V.
Diakenhuisweg 29-35
2033 AP Haarlem
The Netherlands
Tel.: +31 23 5511512

AIMMS Inc.
11711 SE 8th Street
Suite 303
Bellevue, WA 98005
USA
Tel.: +1 425 458 4024

AIMMS Pte. Ltd.
55 Market Street #10-00
Singapore 048941
Tel.: +65 6521 2827

AIMMS
SOHO Fuxing Plaza No.388
Building D-71, Level 3
Madang Road, Huangpu District
Shanghai 200025
China
Tel.: ++86 21 5309 8733

Email: info@aimms.com
WWW: www.aimms.com

This documentation was typeset by AIMMS B.V. using LATEX and the LUCIDA font family.

Part II

Non-Procedural Language Components

Chapter 3

Set Declaration

This chapter

This chapter covers all aspects associated with the declaration and use of sets in AIMMS models. The main topics are indexing with sets, simple sets with strings, simple sets with integers, relations, compound sets and indexed sets.

3.1 Sets and indices

General

Sets and indices give your AIMMS model dimension and depth by providing a mechanism for grouping parameters, variables, and constraints. Sets and indices are also used as driving mechanism in arithmetic operations such as summation. The use of sets for indexing expressions helps to describe large models in a concise and understandable way.

Example

Consider a set of *Cities* and an identifier called *Transport* defined between several pairs of cities (i, j), representing the amount of product transported from supply city i to destination city j. Suppose that you are interested in the quantities arriving in each city. Rather than adding many individual terms, the following mathematical notation, using sets and indices, concisely describes the desired computation of these quantities.

$$(\forall j \in \mathit{Cities}) \qquad \mathit{Arrival}_j = \sum_{i \in \mathit{Cities}} \mathit{Transport}_{ij}.$$

This multidimensional index notation forms the foundation of the AIMMS modeling language, and can be used in all expressions. In this example, i and j are indices that refer to individual *Cities*.

Several types of sets

A set in AIMMS

- has either *strings* or *integers* as elements,
- is either a *simple* set, a *relation*, or a *compound* set, and
- is either *indexed* or *not indexed*.

String versus integer

Sets can either have strings as elements (such as the set *Cities* discussed above), or have integers as elements. An example of an integer set could be a set of *Trials* represented by the numbers $1, \ldots, n$. The resulting integer set can then be used to refer to the results of each single experiment.

Simple versus relation

A *simple* set is a one-dimensional set, such as the set *Cities* mentioned above, while a *relation* or multidimensional set is the Cartesian product of a number of simple sets or a subset thereof. An example of a relation is the set of possible *Routes* between supply and destination cities, which can be represented as a subset of the Cartesian product *Cities* $\times$ *Cities*.

Indexing as basic mechanism

Sets in Aimms are the basis for creating multidimensional identifiers in your model. Through indices into sets you have access to individual values of these identifiers for each tuple of elements. In addition, the indexing notation in Aimms is your basic mechanism for expressing iterative operations such as repeated addition, repeated multiplication, sequential search for a maximum or minimum, etc.

Compound sets

As you shall see, you can use both simple and relations for indexing. If you think, for example, that it is more convenient to express your model components in terms of an index r into the relation *Routes* rather than in terms of tuples (i, j) of cities, Aimms allows you to do so. In Aimms, *compound sets* are a special type of relations, namely those which have an associated index.

Indexed sets

Both simple and compound sets may be indexed. An indexed set is a family of sets defined for every element in the index domain of the indexed set. An example of an indexed set is the set of transport destination cities defined for each supply city. On the other hand, the set *Cities* discussed above is not an indexed set.

Sorting of sets

The contents of any simple or compound set can be sorted in Aimms. Sorting can take place either automatically or manually. Automatic sorting is based on the value of some expression defined for all elements of the set. By using an index into a sorted subset, you can access any subselection of data in the specified order. Such a subselection may be of interest in your end-user interface or at a certain stage in your model.

3.2 Set declaration and attributes

Set attributes

Each set has an optional list of attributes which further specify its intended behavior in the model. The attributes of sets are given in Table 3.1. The attributes `IndexDomain` and `Tags` are only relevant to indexed sets and compound sets, respectively.

Attribute	**Value-type**	**See also page**
`IndexDomain`	*index-domain*	39
`SubsetOf`	*subset-domain*	
`Index`	*identifier-list*	
`Parameter`	*identifier-list*	
`Tags`	*tags*	37
`Text`	*string*	19
`Comment`	*comment string*	19
`Property`	NoSave	
`Definition`	*set-expression*	
`OrderBy`	*expression-list*	

Table 3.1: Set attributes

3.2.1 Simple sets

Definition

A *simple* set in AIMMS is a finite collection of elements. These elements are either strings or integers. Strings are typically used to identify real-world objects such as products, locations, persons, etc.. Integers are typically used for algorithmic purposes. With every simple set you can associate indices through which you can refer (in succession) to all individual elements of that set in indexed statements and expressions.

Most basic example

An example of the most basic declaration for the set *Cities* from the previous example follows.

```
Set Cities {
    Index      : i,j;
}
```

This declares the identifier `Cities` as a simple set, and binds the identifiers `i` and `j` as indices to `Cities` throughout your model text.

More detailed example

Consider a set *SupplyCities* which is declared as follows:

```
Set SupplyCities {
    SubsetOf   : Cities;
    Parameter  : LargestSupplyCity;
    Text       : The subset of cities that act as supply city;
    Definition : {
        { i | Exists( j | Transport(i,j) ) }
    }
    OrderBy    : i;
}
```

The "|" operator used in the definition is to be read as "such that" (it is explained in Chapter 5). Thus, `SupplyCities` is defined as the set of all cities from which there is transport to at least one other city. All elements in the set are ordered lexicographically. The set has no index of its own, but does have an element parameter `LargestSupplyCity` that can hold any particular element with a specific property. For instance, the following assignment forms one way to specify the value of this element parameter:

```
LargestSupplyCity := ArgMax( i in SupplyCities, sum( j, Transport(i,j) ) );
```

Note that this assignment selects that particular element from the subset of `SupplyCities` for which the total amount of `Transport` leaving that element is the largest.

The `SubsetOf` attribute

With the `SubsetOf` attribute you can tell AIMMS that the set at hand is a subset of another set, called the *subset domain*. For simple sets, such a subset domain is denoted by a single set identifier. During the execution of the model AIMMS will assert that this subset relationship is satisfied at all times.

Root sets

Each simple set that is not a subset of another set is called a *root set*. As will be explained later on, root sets have a special role in AIMMS with respect to data storage and ordering.

The `Index` attribute

An index takes the value of *all* elements of a set successively and in the order specified by its declaration. It is used in operations like summation and indexed assignment over the elements of a set. With the `Index` attribute you can associate identifiers as indices into the set at hand. The index attributes of all sets must be unique identifiers, i.e. every index can be declared only once.

The `Parameter` attribute

A parameter declared in the `Parameter` attribute of a set takes the value of a *specific* element of that set. Throughout the sequel we will refer to such a parameter as an *element parameter*. It is a very useful device for referring to set elements that have a special meaning in your model (as illustrated in the previous example). In a later chapter you will see that an element parameter can also be defined separately as a parameter which has a set as its range.

The `Text` and `Comment` attributes

With the `Text` attribute you can specify one line of descriptive text for the end-user. This description can be made visible in the graphical user interface when the data of an identifier is displayed in a page object. You can use the `Comment` attribute to provide a longer description of the identifier at hand. This description is intended for the modeler and cannot be made visible to an end-user. The `Comment` attribute is a multi-line string attribute.

Quoting identifier names in `Comment`

You can make Aimms aware that specific words in your comment text are intended as identifier names by putting them in single quotes. This has the advantage that Aimms will update your comment when you change the name of that identifier in the model editor, or, that Aimms will warn you when a quoted name does not refer to an existing identifier.

The `OrderBy` attribute

With the `OrderBy` attribute you can indicate that you want the elements of a certain set to be ordered according to a single or multiple ordering criteria. Both simple and compound sets can be ordered.

Ordering root sets

A special word of caution is in place with respect to specifying an ordering principle for root sets. Root sets play a special role within Aimms because all data defined over a root set or any of its subsets is stored in the original *data entry* order in which elements have been added to that root set. Thus, the data entry order defines the natural order of execution over a particular domain, and specifying the `OrderBy` attribute of a root set may influence overall execution times of your model in a negative manner. Section 13.2.7 discusses these efficiency aspects in more detail, and provides alternative solutions.

Ordering criteria

The value of the `OrderBy` attribute can be a comma-separated list of one or more ordering criteria. The following ordering criteria (numeric, string or user-defined) can be specified.

- If the value of the `OrderBy` attribute is an indexed numerical expression defined over the elements of the set, Aimms will order its elements in increasing order according to the numerical values of the expression.
- If the value of the `OrderBy` attribute is either an index into the set, a set element-valued expression, or a string expression over the set, then its elements will be ordered lexicographically with respect to the strings associated with the expression. By preceding the expression with a minus sign, the elements will be ordered reverse lexicographically.
- If the value of the `OrderBy` attribute is the keyword `User`, the elements will be ordered according to the order in which they have been added to the subset, either by the user, the model, or by means of the `Sort` operator.

Specifying multiple criteria

When applying a single ordering criterion, the resulting ordering may not be unique. For instance, when you order according to the size of transport taking place from a city, there may be multiple cities with equal transport. You may want these cities to be ordered too. In this case, you can enforce a more refined ordering principle by specifying multiple criteria. Aimms applies all criteria in succession, and will order only those elements that could not be uniquely distinguished by previous criteria.

Example

The following set declarations give examples of various types of automatic ordering. In the last declaration, the cities with equal transport are placed in a lexicographical order.

```
Set LexicographicSupplyCities {
    SubsetOf  : SupplyCities;
    OrderBy   : i;
}
Set ReverseLexicographicSupplyCities {
    SubsetOf  : SupplyCities;
    OrderBy   : - i;
}
Set SupplyCitiesByIncreasingTransport {
    SubsetOf  : SupplyCities;
    OrderBy   : sum( j, Transport(i,j) );
}
Set SupplyCitiesByDecreasingTransportThenLexicographic {
    SubsetOf  : SupplyCities;
    OrderBy   : - sum( j, Transport(i,j) ), i;
}
```

The `Property` attribute

In general, you can use the `Property` attribute to assign additional properties to an identifier in your model. The applicable properties depend on the identifier type. Sets, at the moment, only support a single property.

- The property `NoSave` specifies that the contents of the set at hand will never be stored in a case file. This can be useful, for instance, for intermediate sets that are necessary during the model's computation, but are never important to an end-user.

Dynamic property selection

The properties selected in the `Property` attribute of an identifier are `on` by default, while the nonselected properties are `off` by default. During execution of your model you can also dynamically change a property setting through the `Property` statement. The `PROPERTY` statement is discussed in Section 8.5.

The `Definition` attribute

If an identifier can be uniquely defined throughout your model by a single expression, you can (and should) use the `Definition` attribute to specify this global relationship. AIMMS stores the result of a `Definition` and recomputes it only when necessary. For sets where a global `Definition` is not possible, you can make assignments in procedures and functions. The value of the `Definition` attribute must be a valid expression of the appropriate type, as exemplified in the declaration

```
Set SupplyCities {
    SubsetOf   : Cities;
    Definition : {
        { i | Exists( j | Transport(i,j) ) }
    }
}
```

3.2.2 Integer sets

Integer sets

A special type of simple set is an integer set. Such a set is characterized by the fact that the value of the SubsetOf attribute must be equal to the predefined set Integers or a subset thereof. Integer sets are most often used for algorithmic purposes.

Usage in expressions

Elements of integer sets can also be used as integer values in numerical expressions. In addition, the result of an integer-valued expression can be added as an element to an integer set. Elements of non-integer sets that represent numerical values cannot be used directly in numerical expressions. To obtain the numerical value of such non-integer elements, you can use the Val function (see Section 5.2.2).

Construction

In order to fill an integer set AIMMS provides the special operator ".." to specify an entire range of integer elements. This powerful feature is discussed in more detail in Section 5.1.1.

Example

The following somewhat abstract example demonstrates some of the features of integer sets. Consider the following declarations.

```
Parameter LowInt {
    Range      : Integer;
}
Parameter HighInt {
    Range      : Integer;
}
Set EvenNumbers {
    SubsetOf    : Integers;
    Index       : i;
    Parameter   : LargestPolynomialValue;
    OrderBy     : - i;
}
```

The following statements illustrate some of the possibilities to compute integer sets on the basis of integer expressions, or to use the elements of an integer set in expressions.

```
    ! Fill the integer set with the even numbers between
    ! LowInt and HighInt. The first term in the expression
    ! ensures that the first integer is even.

    EvenNumbers := { (LowInt + mod(LowInt,2)) .. HighInt by 2 };

    ! Next the square of each element i of EvenNumbers is added
    ! to the set, if not already part of it (i.e. the union results)

    for ( i | i <= HighInt ) do
        EvenNumbers += i^2;
```

```
endfor;

! Finally, compute that element of the set EvenNumbers, for
! which the polynomial expression assumes the maximum value.

LargestPolynomialValue := ArgMax( i, i^4 - 10*i^3 + 10*i^2 - 100*i );
```

Ordering integer sets

By default, integer sets are ordered according to the numeric value of their elements. Like with ordinary simple sets, you can override this default ordering by using the `OrderBy` attribute. When you use an index in specifying the order of an integer set, AIMMS will interpret it as a numeric expression.

3.2.3 Relations and compound sets

Relation

A *relation* or multidimensional set is the Cartesian product of a number of simple sets or a subset thereof. Relations are typically used as the domain space for multidimensional identifiers.

Tuples and index components

An element of a relation is called a *tuple* and is denoted by the usual mathematical notation, i.e. as a parenthesized list of comma-separated elements. Throughout, the word *index component* will be used to denote the index of a particular position inside a tuple.

Index tuple

To reference an element in a relation, you can use an *index tuple*, in which each tuple component contains an index corresponding to a simple set.

Compound set

Like simple sets, the elements of a relation can be referenced using a single index, by associating a *compound index* with the relation. In AIMMS, such a special type of relation is called a *compound set*, and, from here on, we will reserve the word *relation* for multidimensional sets which

- do not have an associated compound index, and
- are not a subset of a compound set.

Compound set attributes

Compound sets support all the attributes of a simple set. In addition, compound sets support the `Tags` attribute, which enables you to access and use the individual components of an element tuple in expressions and statements.

The `SubsetOf` attribute

The `SubsetOf` attribute is mandatory for both relations and compound sets, and must contain the *subset domain* of the set. This subset domain is denoted either as a parenthesized comma-separated list of simple set identifiers, or, if it is a subset of another relation or compound set, just the name of that set.

Example

The following example demonstrates some elementary declarations of relations and compound sets, given the two-dimensional parameters `Distance(i,j)` and `TransportCost(i,j)`. The following set declaration defines a relation.

```
Set HighCostConnections {
    SubsetOf   : (Cities, Cities);
    Definition : {
        { (i,j) | Distance(i,j) > 0 and TransportCost(i,j) > 100 }
    }
}
```

The following two declarations define compound sets, either because they have an associated compound index, or are a subset of a compound set.

```
Set ConnectedCities {
    SubsetOf   : (Cities, Cities);
    Index      : cc;
    Definition : {
        { (i,j) | Distance(i,j) > 0 }
    }
}
Set ExpensiveConnections {
    SubsetOf   : ConnectedCities;
    Definition : {
        { cc | TransportCost(cc) > 100 }
    }
}
```

Accessing compound set elements

The above example shows that you can specify indices and element parameters for compound sets just as you could for simple sets. They are called *compound indices* and *compound element parameters*. As long as the domains are compatible, AIMMS lets you unrestrictedly interchange compound indices with tuples of indices as illustrated in the definitions of `ConnectedCities` and `ExpensiveConnections`.

The Tags attribute

When you use compound indices or compound element parameters, you still have access to individual index components through the use of *tags*, which can be declared in the `Tags` attribute of a compound set. Tags provide a placeholder for every individual component of a tuple in a compound set to be used in expressions and assignments. To reference a specific index component or subtuple, one can simply use the fairly standard dot notation followed by a single tag or a tuple of tags.

Examples

Consider an extension of the declaration of the set ConnectedCities with a Tags attribute as follows.

```
Set ConnectedCities {
    SubsetOf   : (Cities, Cities);
    Index      : cc;
    Tags       : (orig, dest);
    Comment    : {
        The tag orig for the first index component stands for originating cities,
        while the second index component dest stands for destination cities.;
    }
}
```

The following compound set declarations demonstrate the use of these tags.

```
Set LinksWithSupplyCities {
    SubsetOf   : ConnectedCities;
    Definition : {
        { cc | cc.orig in SupplyCities }
    }
}
Set BidirectionalLinks {
    SubsetOf   : ConnectedCities;
    Definition : {
        { cc | cc.(dest,orig) in ConnectedCities }
    }
}
```

Note that the second example above reverses the tag order. As a result, only those connections that also have a reverse link are selected.

Uniqueness

The tag names that you specify in the Tags attribute are Aimms identifiers, and hence must be unique within your application. The length of a tag tuple (i.e. the number of components) must coincide with the dimension of the compound set. Within the tree of a compound root set and all of its subsets, you can only specify a tuple of tag names for the root set. However, you are allowed to use these tags for all subsets in the entire tree.

Ordering compound sets

By default, compound sets are ordered componentwise from left to right, and consistent with the ordering of the underlying simple root sets. By using one or more ordering criteria, you can overrule this default ordering, as illustrated by the following declaration.

Example

```
Set LinksByDistanceThenDoublyLexicographic {
    SubsetOf   : ConnectedCities;
    Definition : ConnectedCities;
    OrderBy    : Distance(cc), cc.orig, cc.dest;
}
```

Relation versus compound set

You may wonder why AIMMS makes such a clear distinction between relations and compound sets. The reason is that relations and compound sets lead to completely different storage and access characteristics.

- When you define or use a relation, for instance, in the domain restriction of a multidimensional parameter, AIMMS still uses multidimensional index tuples to refer to the elements in the relation.
- When you use a compound index in the declaration of a multidimensional parameter, AIMMS considers that index to refer to a simple set consisting of compound elements, and you need *tags* to refer to the individual components of the compound elements.

Reduced memory versus increased access time

The use of compound sets will reduce the effective dimension of the multidimensional data in your model, possibly leading to a dramatically lower memory usage for very high-dimensional identifiers. However, you should also keep in mind that accessing the individual components of a compound element may become much more expensive in the presence of compound indices. Thus, whether or not to use compound sets is a trade-off which needs careful investigation.

Not all operations supported

In addition, in deciding to use compound set, you should take into consideration that reading and writing of compound sets, and data declared over compound sets, from and to databases is not supported (see also Chapter 27). However, in such cases it is still possible to read in the underlying relation, and convert the data to a compound set representation by assigning the relation to the compound set.

3.2.4 Indexed sets

Definition

An *indexed set* represents a family of sets defined for all elements in another set, called the *index domain*. The elements of all members of the family must be from a single (sub)set. Although membership tables allow you to reach the same effect, indexed sets often make it possible to express certain operations very concisely and intuitively.

The `IndexDomain` attribute

A set becomes an indexed set by specifying a value for the `IndexDomain` attribute. The value of this attribute must be a single index or a tuple of indices, optionally followed by a logical condition. The precise syntax of the `IndexDomain` attribute is discussed on page 44.

Example

The following declarations illustrate some indexed sets with a content that varies for all elements in their respective index domains.

```
Set SupplyCitiesToDestination {
    IndexDomain  : j;
    SubsetOf     : Cities;
    Definition   : {
        { i | Transport(i,j) }
    }
}
Set DestinationCitiesFromSupply {
    IndexDomain  : i;
    SubsetOf     : Cities;
    Definition   : {
        { j | Transport(i,j) }
    }
}
Set IntermediateTransportCities {
    IndexDomain  : (i,j);
    SubsetOf     : Cities;
    Definition   : DestinationCitiesFromSupply(i) * SupplyCitiesToDestination(j);
    Comment      : {
        All intermediate cities via which an indirect transport
        from city i to city j with one intermediate city takes place
    }
}
```

The first two declarations both define a one-dimensional family of subsets of `Cities`, while the third declaration defines a two-dimensional family of subsets of `Cities`. Note that the `*` operator is applied to sets, and therefore denotes intersection.

Subset domains

The subset domain of an indexed set family can be either a simple set identifier, a compound set identifier, or another family of indexed simple or compound sets of the same or lower dimension. The subset domain of an indexed set *cannot* be a relation.

No default indices

Declarations of indexed sets do not allow you to specify either the `Index` or `Parameter` attribute. Consequently, if you want to use an indexed set for indexing, you must locally bind an index to it. For more details on the use of indices and index binding refer to Sections 3.3 and 9.1.

3.3 `INDEX` declaration and attributes

Direct versus indirect declaration

Every index used in your model must be declared exactly once. You can declare indices *indirectly*, through the `Index` attribute of a simple or compound set, or *directly* using an `Index` declaration. Note that all previous examples show indirect declaration of indices.

`Index` declaration

When you choose to declare an index not as an attribute of a set declaration, you can use the `Index` declaration. The attributes of each single index declaration are given in Table 3.2.

Attribute	**Value-type**	**See also page**
`Range`	*set-identifier*	
`Text`	*string*	19
`Comment`	*comment string*	19

Table 3.2: `Index` attributes

The `Range` attribute

You can assign a default binding with a specific set to directly declared indices by specifying the `Range` attribute. If you omit this `Range` attribute, the index has no default binding to a specific set and can only be used in the context of local or implicit index binding. The details of index binding are discussed in Section 9.1.

Example

The following declaration illustrates a direct `Index` declaration.

```
Index c {
    Range : Customers;
}
```