

---

## **AIMMS Function Reference - Set Functions**

This file contains only one chapter of the book. For a free download of the complete book in pdf format, please visit [www.aimms.com](http://www.aimms.com)

## Set Related Functions

AIMMS supports the following set related functions:

- `ActiveCard`
- `Card`
- `CloneElement`
- `ConstraintVariables`
- `Element`
- `ElementCast`
- `ElementRange`
- `FindUsedElements`
- `First`
- `Last`
- `Ord`
- `RestoreInactiveElements`
- `RetrieveCurrentVariableValues`
- `SetAddRecursive`
- `SetElementAdd`
- `SetElementRename`
- `StringToElement`
- `SubRange`
- `VariableConstraints`

---

## ActiveCard

The function `ActiveCard` returns the cardinality of active elements in its identifier argument, or the cardinality of active elements of a suffix of that identifier.

```
Card(  
  Identifier,    ! (input) identifier reference  
  [Suffix]      ! (optional) element in the set AllSuffixNames  
)
```

### Arguments:

#### *Identifier*

A reference to a set or an indexed identifier.

#### *Suffix*

An element in the predefined set of `AllSuffixNames`.

### Return value:

If *Identifier* is a set, the function `ActiveCard` returns the number of active elements in *Identifier*. If *Identifier* is an indexed identifier, the function `ActiveCard` returns the number of nondefault values stored for *Identifier*. If *Suffix* is given, the number of nondefault values stored for the given suffix of *Identifier*.

### Remarks:

The `ActiveCard` function cannot be applied to slices of indexed identifiers. In such a case, you can use the `Count` operator to count the number of nondefault elements.

### See also:

The function `Card` and `Count` operator (see also Section 6.1.6 of the Language Reference). The set `AllSuffixNames`.

---

## Card

The function `Card` returns the cardinality of its identifier argument, or of the cardinality of a suffix of that identifier.

```
Card(
  Identifier,    ! (input) identifier reference
  [Suffix]      ! (optional) element in the set AllSuffixNames
)
```

### Arguments:

#### *Identifier*

A reference to a set or an indexed identifier.

#### *Suffix*

An element in the predefined set of `AllSuffixNames`.

### Return value:

If *Identifier* is a set, the function `Card` returns the number of elements in *Identifier*. If *Identifier* is an indexed identifier, the function `Card` returns the number of nondefault values stored for *Identifier*. If *Suffix* is given, the number of nondefault values stored for the given suffix of *Identifier*.

### Remarks:

- The `Card` function cannot be applied to slices of indexed identifiers. In such a case, you can use the `Count` operator to count the number of nondefault elements.
- When the `Card` function is used inside the definition of a parameter or a set and the first argument is an index or element parameter into the set `AllIdentifiers` then the definition depends on all identifiers that can appear on the left hand side of an assignment (sets without a definition, parameters without a definition, variables and constraints). The cardinality will be computed for all identifiers, including those with a definition. These definitions will not be made up to date, however. This is illustrated in the following example.

```
PARAMETER:
  identifier : A ;
PARAMETER:
  identifier : B
  definition : A + 1 ;
PARAMETER:
  identifier : TheCards
  index domain : IndexIdentifiers
  definition : Card( IndexIdentifiers, 'Level' );
body:
  A := 1 ;
  display TheCards ;
```

Here `TheCards` is computed in the display statement because `A` just changed. The definition of `TheCards`, that is made up to date by the display statement, will, however, not invoke the computation of `B`, although it is not up to date. This is done in order to avoid circular references while making set and parameter definitions up to date. In order to make `B` up to date consider using the `Update` statement, see also Section 7.3 of the Language Reference.

**See also:**

The function `ActiveCard` and the `Count` operator (see also Section 6.1.6 of the Language Reference). The set `AllSuffixNames`.

---

## CloneElement

The procedure `CloneElement` copies the data associated with a particular element to another element.

```
CloneElement(
  updateSet,           ! (input, output) a set identifier
  originalElement,    ! (input) an element in the set
  cloneName,          ! (input) a string that is the name of the clone
  cloneElement,       ! (output) an element parameter
  includeDefinedSubsets ) ! (optional) an integer, default 0.
```

The procedure `CloneElement` performs the following actions:

1. It creates or finds an element with name `cloneName`: `cloneElement`. The element `cloneElement` is inserted into `updateSet` if it is not already there. This insertion is only permitted if `updateSet` does not have a definition.
2. For each domain set of `updateSet`, say `insertDomainSet`, the element `cloneElement` is inserted into `insertDomainSet` if it is not already there. Such an insertion is only permitted if `insertDomainSet` does not have a definition.
3. For each subset of `updateSet`, say `insertSubset` in which `originalElement` is an element, `cloneElement` is also inserted into `insertSubset`. If `includeDefinedSubsets` is 0, then `insertSubset` is skipped if it is a defined subset.
4. The domain sets of steps 1 and 2, and the sets modified in step 3 form a set, say `modifiedSets`.
5. Identifiers declared over a set in `modifiedSets` that meet one of the following criteria, are selected:
  - It is a non-local multi-dimensional set without a definition.
  - It is a non-local parameter without a definition.
  - It is a variable.
  - It is a constraint.
 These identifiers form the set `modifiedIdentifiers`.
6. For each identifier in the set `modifiedIdentifiers`, and all suffices of this identifier, the data associated with element `originalElement` is copied to `cloneElement`.

### Arguments:

*updateSet*

A one-dimensional non-compound set.

*originalElement*

An element valued expression that should result in an element in `updateSet`.

*cloneName*

A string expression that should result in a name that is in the set `updateSet` or can be added to that set.

*cloneElement*

An element parameter, in which the resulting element is stored.

*includeDefinedSubsets*

When non-zero, defined subsets are included in the modifiedSets as well. When these defined subsets are evaluated thereafter again, this may result in the creation of inactive data. Inactive data can be removed by a CLEANUP or CLEANDEPENDENTS statement, see Section 23.3 of the language reference. Defined subsets that are defined as an enumeration are never included.

**Return value:**

The procedure returns 1 if successful and 0 otherwise. Possible reasons for returning 0 are:

- originalElement is not in updateSet.
- cloneName equals name of originalElement.
- There are no identifiers modified.

**Remarks:**

If you want to make sure that the string cloneName is not yet an element in updateSet, use a statement like:

```
if ( not ( cloneName in updateSet ) ) then
  CloneElement( ... );
endif ;
```

**Example:**

With the following declarations (and initial data):

```
SET:
  identifier  : S
  index      : i, j
  parameter  : ep
  initial data : data { a } ;

PARAMETER:
  identifier  : P
  index domain : i
  initial data : data { a : 1 } ;

PARAMETER:
  identifier  : Q
  index domain : (i,j)
  initial data : data { ( a, a ) : 1 } ;
```

the statement

```
CloneElement( S, 'a', "b", ep );
```

results in S, P, Q and ep having the following data:

```
S := data { a, b } ;  
P := data { a : 1, b : 1 } ;  
Q := data { ( a, a ) : 1, ( a, b ) : 1, ( b, a ) : 1, ( b, b ) : 1 } ;  
ep := 'b' ;
```

**See also:**

The function [StringToElement](#), the procedure [FindUsedElements](#) and the procedure [RestoreInactiveElements](#).

---

## ConstraintVariables

The function `ConstraintVariables` returns all the symbolic variables that are referred in a certain collection of constraints, including the variables that are referred in the definitions of these variables.

```
ConstraintVariables(  
  Constraints      ! (input) a subset of AllConstraints  
)
```

### Arguments:

*Constraints*

The set of constraints for which you want to retrieve the referred variables.

### Return value:

The function returns a subset of the set `AllVariables`, containing the variables found.

### See also:

The function [VariableConstraints](#).

---

## Element

With the function `Element` you can retrieve the  $n$ -th element from a set.

```
Element(  
    Set,          ! (input) set reference  
    n            ! (input) integer expression  
)
```

### Arguments:

*Set*

The set from which an element is to be returned.

$n$

An integer expression indicating the ordinal number of the element to be returned.

### Return value:

The function `Element` returns the  $n$ -th element of set *Set*.

### Remarks:

If there is no  $n$ -th element in *Set*, the function returns the empty element '' instead.

---

## ElementCast

With the function `ElementCast` you can cast an element of one set to an (existing) element with the same name in a set with a different root set.

```
ElementCast(  
  set,           ! (input) a set expression  
  element,      ! (input) a scalar element expression  
  [create]     ! (optional) 0 or 1  
)
```

### Arguments:

*set*

A set in which you want to find a specific element name.

*element*

A scalar element expression, representing the element that you want to convert to a different root set hierarchy.

*create (optional)*

An indicator whether or not a nonexisting element are added to the set during the call.

### Return value:

The function returns the existing element or, if the element cannot be converted to an existing element and the argument *create* is not set to 1, the function returns the empty element. If *create* is set to 1, nonexisting elements will be created on the fly.

### See also:

The procedure `SetElementAdd`.

---

## ElementRange

With the function `ElementRange` you can create a set with elements in which each element can be constructed using a prefix string, a postfix string, and a sequential number.

```
ElementRange(  
  from,          ! (input) integer expression  
  to,           ! (input) integer expression  
  [incr,]       ! (optional) integer expression  
  [prefix,]     ! (optional) string expression  
  [postfix,]    ! (optional) string expression  
  [fill]        ! (optional) 0 or 1  
)
```

### Arguments:

*from*

The integer value for which the first element must be created

*to*

The integer value for which the last element must be created

*incr (optional)*

The integer-valued interval length between two consecutive elements. If omitted, then the default interval length of 1 is used.

*prefix (optional)*

The prefix string for every element. If omitted, then the elements have no prefix (and thus start with the number).

*postfix (optional)*

The postfix string for every element. If omitted, then the elements have no postfix (and thus end with the number).

*fill (optional)*

This logical indicator specifies whether the numbers must be padded with leading zeroes. If omitted, then the default value 1 is used.

### Return value:

The function returns a set containing the created elements.

---

## FindUsedElements

The procedure FindUsedElements finds all elements of a particular set that are in use in a given collection of indexed model identifiers.

```
FindUsedElements(  
  SearchSet,      ! (input) a set  
  SearchIdentifiers, ! (input) a subset of AllIdentifiers  
  UsedElements   ! (output) a subset  
)
```

### Arguments:

#### *SearchSet*

The set for which you want to find the used elements.

#### *SearchIdentifiers*

A subset of AllIdentifiers, holding identifiers that are indexed over SearchSet.

#### *UsedElements*

A subset of *SearchSet*. On return this subset will contain the elements that are currently used (i.e. have corresponding nondefault values) in the identifiers contained in *SearchIdentifiers*.

---

**First**

With the function `First` you can retrieve the first element from a set.

```
First(  
    Set,          ! (input) set reference  
)
```

**Arguments:**

*Set*  
The set from which the first element is to be returned.

**Return value:**

The function `First` returns the first element of set *Set*.

**Remarks:**

If there is no element in *Set*, the function returns the empty element '' instead.

---

**Last**

With the function Last you can retrieve the last element from a set.

```
Last(  
    Set,          ! (input) set reference  
)
```

**Arguments:**

*Set*

The set from which the last element is to be returned.

**Return value:**

The function Last returns the last element of set *Set*.

**Remarks:**

If there is no element in *Set*, the function returns the empty element '' instead.

---

## Ord

The function `Ord` returns the ordinal number of a set element relative to a set.

```
Ord(  
  index,      ! (input) element expression  
  [set]      ! (optional) set reference  
)
```

### Arguments:

*index*

An element expression for which you want to obtain the ordinal number.

*set (optional)*

The set with respect to which you want the ordinal number to be taken. If omitted, *set* is assumed to be the range of the argument *index*.

### Return value:

The function `Ord` returns the ordinal number of *index* in set *set*.

### Remarks:

A compile time error occurs if the argument *set* is not present, and AIMMS is unable to determine the range of *index*.

---

## RestoreInactiveElements

The procedure `RestoreInactiveElements` finds and restores all elements that were previously removed from a particular set, but for which inactive data still exists in a given collection of indexed model identifiers.

```
RestoreInactiveElements(  
  SearchSet,      ! (input/output) a set  
  SearchIdentifiers, ! (input) a subset of AllIdentifiers  
  UsedElements    ! (output) a subset  
)
```

### Arguments:

#### *SearchSet*

The set for which you want to find the inactive elements.

#### *SearchIdentifiers*

A subset of *AllIdentifiers*, holding identifiers that are indexed over *SearchSet*.

#### *UsedElements*

A subset of *SearchSet*. On return this subset will contain all the inactive elements that are currently used (i.e. have corresponding nondefault values) in the identifiers contained in *SearchIdentifiers*.

### Remarks:

The inactive elements found are placed in the *result-set*, but are also automatically added to the *search-set*.

---

## RetrieveCurrentVariableValues

With the procedure `RetrieveCurrentVariableValues` you can obtain the variable values for a given collection of variables during a running solution process. This procedure can only be called from within the context of a solver callback procedure.

```
RetrieveCurrentVariableValues(  
    Variables    ! (input) a subset of AllVariables  
)
```

### Arguments:

#### *Variables*

A subset of `AllVariables`, holding all the variables for which you want to retrieve the current values.

### See also:

Solver callback procedures are discussed in full detail in [Section 15.2](#) of the Language Reference

---

## SetAddRecursive

With the procedure SetAddRecursive you can merge the elements of one set into another set.

```
SetAddRecursive(  
    toSet,          ! (input/output) a set  
    fromSet        ! (input) a set  
)
```

### Arguments:

*toSet*

The set into which the elements of *fromSet* are merged.

*fromSet*

The set that you want to merge in *toSet*.

### Remarks:

- The sets *toSet* and *fromSet* should have the same root set.
- The difference between this function and a regular set assignment is that in case *fromSet* is not the domain of *toSet* all elements added to *toSet* will also be added to the domain set of *toSet*

---

## SetElementAdd

With the procedure `SetElementAdd` you can add new elements to a set. When you apply `SetElementAdd` to a root set, the element will be added to that root set. When you apply it to a subset, the element will be added to the subset as well as to all its supersets, up to and including its associated root set.

```
SetElementAdd(  
    set,           ! (input/output) a set  
    new-element,  ! (output) an element parameter  
    element-name  ! (input) a scalar string expression  
)
```

### Arguments:

*set*

The root set or subset to which you want to add the element.

*new-element*

An element parameter into *set*, that on return will point to the newly added element.

*element-name*

A string holding the name of the element to be added.

### Remarks:

If the element already exists in the set, the procedure does not make any changes to the set, and on return the element parameter *new-element* will point to the existing element.

### See also:

The function `ElementCast`. The procedures `SetElementRename`, `StringToElement`.

---

## SetElementRename

With the procedure `SetElementRename` you can rename an element in a set.

```
SetElementRename(  
    Setname,      ! (input) a set  
    Element,      ! (input) an element  
    Newname       ! (input) a scalar string expression  
)
```

### Arguments:

*Setname*

The root set or subset in which you want to rename an element.

*Element*

The element that you want to rename.

*Newname*

A string holding the new name of the element.

### Remarks:

- If the new name for the element already exists in the set, the procedure will generate an execution error.
- AIMMS will refuse to rename a set element, if an explicit reference to such an element exists in the model source.

### See also:

The procedure `SetElementAdd`. The function `StringToElement`.

---

## StringToElement

With the function `StringToElement` you can convert a string into an (existing) element of a set.

```
StringToElement(  
  Set,          ! (input) a set expression  
  Name,        ! (input) a scalar string  
  [create]     ! (optional) 0 or 1, default 0  
)
```

### Arguments:

*Set*

A set in which you want to find a specific element name.

*Name*

A scalar string expression, representing the string that you want to convert.

*create (optional)*

An indicator whether or not a nonexisting element are added to the set during the call.

### Return value:

The function returns the existing element or, if the string cannot be converted to an existing element and the argument *create* is not set to 1, the function return the empty element. If *create* is set to 1, nonexisting elements will be created on the fly.

### See also:

The function `ElementCast`. The procedure `SetElementAdd`.

---

## SubRange

The function `SubRange` extracts a subrange of consecutive elements from an existing set.

```
SubRange(  
  Superset,    ! (input) a simple or compound set  
  First,      ! (input) an element  
  Last        ! (input) an element  
)
```

### Arguments:

#### *Superset*

The set containing the subrange of elements that you want to extract.

#### *First*

An element in *Superset* representing the first element of the subrange.

#### *Last*

An element in *Superset* representing the last element of the subrange.

### Return value:

The function returns a set containing the subrange of elements extracted from *Superset*. If the element *First* is positioned after *Last*, then the empty set is returned.

---

## VariableConstraints

The function `VariableConstraints` returns all the symbolic constraints that refer to one or more variables in a given set of variables.

```
VariableConstraints(  
    Variables ! (input) a subset of AllVariables  
)
```

### Arguments:

*Variables*

The set of variables for which you want to retrieve the constraints that refer to them.

### Return value:

The function returns a subset of the set `AllConstraints`, containing the constraints found.

### See also:

The function [ConstraintVariables](#).