A

AMPL Reference Manual

AMPL is a language for algebraic modeling and mathematical programming: a computer-readable language for expressing optimization problems such as linear programming in algebraic notation. This appendix summarizes the features of AMPL, with particular emphasis on technical details not fully covered in the preceding chapters. Nevertheless, not every feature, construct, and option is listed; the AMPL web site `www.ampl.com` contains the most up to date and complete information.

The following notational conventions are used. Literal text is printed in `constant width font`, while syntactic categories are printed in *italic font.* Phrases or subphrases enclosed in slanted square brackets *[* and *]* are optional, as are constructs with the subscript *opt*.

A.1 Lexical rules

AMPL models involve variables, constraints, and objectives, expressed with the help of sets and parameters. These are called *model entities.* Each model entity has an alphanumeric name: a string of one or more Unicode UTF-8 letters, digits, and underscores, in a pattern that cannot be mistaken for a numeric constant. Upper-case letters are distinct from lower-case letters.

Numeric constants are written in standard scientific notation: an optional sign, a sequence of digits that may contain a decimal point, and an optional exponent field that begins with one of the letters `d`, `D`, `e` or `E`, as in `1.23D-45`. All arithmetic in AMPL is in the same precision (double precision on most machines), so all exponent notations are synonymous.

Literals are strings delimited either by single quotes `'` or by double quotes `"`; the delimiting character must be doubled if it appears within the literal, as in `'x''y'`, which is a literal containing the three characters `x'y`. Newline characters may appear within a literal only if preceded by `\`. The choice of delimiter is arbitrary; `'abc'` and `"abc"` denote the same literal.

Literals are distinct from numeric constants: `1` and `'1'` are unrelated.

Input is free form; white space (any sequence of space, tab or newline characters) may appear between any tokens. Each statement ends with a semicolon.

Comments begin with `#` and extend to the end of the current line, or are delimited by `/*` and `*/`, in which case they may extend across several lines and do not nest. Comments may appear anywhere in declarations, commands, and data.

These words are reserved, and may not be used in other contexts:

```
Current         complements     integer          solve_result_num
IN              contains        less             suffix
INOUT           default         logical          sum
Infinity        dimen           max              symbolic
Initial         div             min              table
LOCAL           else            option           then
OUT             environ         setof            union
all             exists          shell_exitcode   until
binary          forall          solve_exitcode   while
by              if              solve_message    within
check           in              solve_result
```

Words beginning with underscore are also reserved. The other keywords, function names, etc., are predefined, but their meanings can be redefined. For example, the word `prod` is predefined as a product operator analogous to `sum`, but it can be redefined in a declaration, such as

```
set prod;  # products
```

Once a word has been redefined, the original meaning is inaccessible.

AMPL provides synonyms for several keywords and operators; the preferred forms are on the left.

```
^           **
=           ==
<>          !=
and         &&
not         !
or          ||
prod        product
```

A.2 Set members

A set contains zero or more elements or members, each of which is an ordered list of one or more components. Each member of a set must be distinct. All members must have the same number of components; this common number is called the set's dimension.

A literal set is written as a comma-separated list of members, between braces `{` and `}`. If the set is one-dimensional, the members are simply numeric constants or literal strings, or any expressions that evaluate to numbers or strings:

```
{"a","b","c"}
{1,2,3,4,5,6,7,8,9}
{t,t+1,t+2}
```

For a multidimensional set, each member must be written as a parenthesized comma-separated list of the above:

```
{("a",2),("a",3),("b",5)}
{(1,2,3),(1,2,4),(1,2,5),(1,3,7),(1,4,6)}
```

The value of a numeric member is the result of rounding its decimal representation to a floating-point number. Numeric members that appear different but round to the same floating-point number, such as `1` and `0.01E2`, are considered the same.

A.3 Indexing expressions and subscripts

Most entities in AMPL can be defined in collections indexed over a set; individual items are selected by appending a bracketed subscript to the name of the entity. The range of possible subscripts is indicated by an *indexing expression* in the entity's declaration. Reduction operators, such as `sum`, also use indexing expressions to specify sets over which operations are iterated.

A subscript is a list of symbolic or numeric expressions, separated by commas and enclosed in square brackets, as in `supply[i]` and `cost[j,p[k]+1,"O+"]`. Each subscripting expression must evaluate to a number or a literal. The resulting value or sequence of values must give a member of a relevant one-dimensional or multidimensional indexing set.

An indexing expression is a comma-separated list of set expressions, followed optionally by a colon and a logical ''such that'' expression, all enclosed in braces:

indexing:
    { *sexpr-list* }
    { *sexpr-list* : *lexpr* }

sexpr-list:
    sexpr
    dummy-member `in` *sexpr*
    sexpr-list , *sexpr*

Each set expression may be preceded by a dummy member and the keyword `in`. A dummy member for a one-dimensional set is an unbound name, that is, a name not currently defined. A dummy member for a multidimensional set is a comma-separated list, enclosed in parentheses, of expressions or unbound names; the list must include at least one unbound name.

A dummy member introduces one or more dummy indices (the unbound names in its components), whose *scopes*, or ranges of definition, begin just after the following *sexpr*; an index's scope runs through the rest of the indexing expression, to the end of the declaration using the indexing expression, or to the end of the operand that uses the indexing expression. When a dummy member has one or more expression components, the dummy indices in the dummy member range over a *slice* of the set, i.e., they assume all values for which the dummy member is in the set.

```
{A}                           # a set
{A, B}                        # all pairs, one from A, one from B
{i in A, j in B}              # the same
{i in A, B}                   # the same
{i in A, C[i]}                # all pairs, one from A, one from C[i]
{i in A, (j,k) in D}          # 1 from A and 1 (itself a pair) from D
{i in A: p[i] > 0}            # all i in A such that p[i] is positive
{i in A, j in C[i]: i <= j}   # i and j must be numeric
{i in A, (i,j) in D: i <= j} # all pairs with i in A and i,j in D
                              # (same value of i) and i <= j
```

The optional `:` *lexpr* in an indexing expression selects only the members that satisfy the logical expression and excludes the others. The *lexpr* typically involves one or more dummy indices of the indexing expression.

A.4 Expressions

Various items can be combined in AMPL's arithmetic and logical expressions. An expression that may not contain variables is denoted *cexpr* and is sometimes called a ''constant expression'',

Precedence	Name	Type	Remarks
1	`if-then-else`	A, S	A: if no `else`, then ‘‘`else 0`’’ assumed S: ‘‘`else` *sexpr*’’ required
2	`or ||`	L	
3	`exists forall`	L	logical reduction operators
4	`and &&`	L	
5	`< <= = == <> != >= >`	L	
6	`in not in`	L	membership in set
6	`within not within`	L	`S within T` means set `S` ⊆ set `T`
7	`not !`	L	logical negation
8	`union diff symdiff`	S	`symdiff` ≡ symmetric difference
9	`inter`	S	set intersection
10	`cross`	S	cross or Cartesian product
11	`setof .. by`	S	set constructors
12	`+ - less`	A	*a* `less` *b* ≡ max(*a* − *b*, 0)
13	`sum prod min max`	A	arithmetic reduction operators
14	`* / div mod`	A	`div` ≡ truncated quotient of integers
15	`+ -`	A	unary plus, unary minus
16	`^ **`	A	exponentiation

Operators are listed in increasing precedence order. Exponentiation and `if-then-else` are right-associative; the other operators are left-associative. The ‘ Type’ column indicates result types: A for arithmetic, L for logical, S for set.

Table A-1: Arithmetic, logical and set operators.

even though it may involve dummy indices. A logical expression, denoted *lexpr*, may not contain variables when it is part of a *cexpr*. Set expressions are denoted *sexpr*.

Table A-1 summarizes the arithmetic, logical and set operators; the type column indicates whether the operator produces an arithmetic value (A), a logical value (L), or a set value (S).

Arithmetic expressions are formed from the usual arithmetic operators, built-in functions, and arithmetic reduction operators like `sum`:

expr:

number	
variable	
expr arith-op expr	*arith-op* is `+ - less * / mod div ^ **`
unary-op expr	*unary-op* is `+ -`
built-in `(` *exprlist* `)`	
`if` *lexpr* `then` *expr* `[ else` *expr* `]`	
reduction-op indexing expr	*reduction-op* is `sum prod max min`
`(` *expr* `)`	

Built-in functions are listed in Table A-2.

The arithmetic reduction operators are used in expressions like

```
sum {i in Prod} cost[i] * Make[i]
```

The scope of the *indexing* expression extends to the end of the *expr*. If the operation is over an empty set, the result is the identity value for the operation: 0 for `sum`, 1 for `prod`, `Infinity` for `min`, and `-Infinity` for `max`.

Logical expressions appear where a value of ''true'' or ''false'' is required: in `check` statements, the ''such that'' parts of indexing expressions (following the colon), and in `if` *lexpr* `then` ... `else` ... expressions. Numeric values that appear in any of these contexts are implicitly coerced to logical values: 0 is interpreted as false, and all other numeric values as true.

lexpr:

expr	
expr compare-op expr	*compare-op* is `< <= = == != <> > >=`
lexpr logic-op lexpr	*logic-op* is `or || and &&`
`not` *lexpr*	
member `in` *sexpr*	
member `not in` *sexpr*	
sexpr `within` *sexpr*	
sexpr `not within` *sexpr*	
opname indexing lexpr	*opname* is `exists` or `forall`
`(` *lexpr* `)`	

The `in` operator tests set membership. Its left operand is a potential set member, i.e., an expression or comma-separated list of expressions enclosed in parentheses, with the number of expressions equal to the dimension of the right operand, which must be a set expression. The `within` operator tests whether one set is contained in another. The two set operands must have the same dimension.

The logical reduction operators `exists` and `forall` are the iterated counterparts of `or` and `and`. When applied over an empty set, `exists` returns false and `forall` returns true.

Set expressions yield sets.

sexpr:

`{` *[member [, member...]]* `}`	
sexpr set-op sexpr	*set-op* is `union diff symdiff inter cross`
opname indexing sexpr	*opname* is `union` or `inter`
expr `..` *expr [* `by` *expr]*	
`setof` *indexing member*	
`if` *lexpr* `then` *sexpr* `else` *sexpr*	
`(` *sexpr* `)`	
interval	
infinite-set	
indexing	

Components of members can be arbitrary constant expressions. Section A.6.3 describes *interval*s and *infinite-set*s.

When used as binary operators, `union` and `inter` denote the binary set operations of union and intersection. These keywords may also be used as reduction operators.

The `..` operator constructs sets. The default `by` clause is `by 1`. In general, e_1 `..` e_2 `by` e_3 means the numbers

$$e_1,\ e_1 + e_3,\ e_1 + 2e_3,\ \ldots,\ e_1 + \left\lfloor \frac{e_2 - e_1}{e_3} \right\rfloor e_3$$

rounded to set members. (The notation $\lfloor x \rfloor$ denotes the floor of x, that is, the largest integer $\leq x$.)

The `setof` operator is a set construction operator; *member* is either an expression or a comma-separated list of expressions enclosed in parentheses. The resulting set consists of all the *member*s obtained by iterating over the *indexing* expression; the dimension of the resulting expression is the number of components in *member*.

`abs(`x`)`	absolute value $\lvert x \rvert$
`acos(`x`)`	inverse cosine, $\cos^{-1}(x)$
`acosh(`x`)`	inverse hyperbolic cosine, $\cosh^{-1}(x)$
`alias(`v`)`	alias of model entity v
`asin(`x`)`	inverse sine, $\sin^{-1}(x)$
`asinh(`x`)`	inverse hyperbolic sine, $\sinh^{-1}(x)$
`atan(`x`)`	inverse tangent, $\tan^{-1}(x)$
`atan2(`y`, `x`)`	inverse tangent, $\tan^{-1}(y/x)$
`atanh(`x`)`	inverse hyperbolic tangent, $\tanh^{-1}(x)$
`ceil(`x`)`	ceiling of x (next higher integer)
`ctime()`	current time as a string
`ctime(`t`)`	time t as a string
`cos(`x`)`	cosine
`exp(`x`)`	e^x
`floor(`x`)`	floor of x (next lower integer)
`log(`x`)`	$\log_e(x)$
`log10(`x`)`	$\log_{10}(x)$
`max(`x`, `y`, ...)`	maximum (2 or more arguments)
`min(`x`, `y`, ...)`	minimum (2 or more arguments)
`precision(`x`, `n`)`	x rounded to n significant decimal digits
`round(`x`, `n`)`	x rounded to n digits past decimal point
`round(`x`)`	x rounded to an integer
`sin(`x`)`	sine
`sinh(`x`)`	hyperbolic sine
`sqrt(`x`)`	square root
`tan(`x`)`	tangent
`tanh(`x`)`	hyperbolic tangent
`time()`	current time in seconds
`trunc(`x`, `n`)`	x truncated to n digits past decimal point
`trunc(`x`)`	x truncated to an integer

Table A-2: Built-in arithmetic functions.

```
ampl: set y = setof {i in 1..5} (i,i^2);
ampl: display y;
set y := (1,1) (2,4) (3,9) (4,16) (5,25);
```

A.4.1 Built-in functions

The built-in arithmetic functions are listed in Table A-2. The function `alias` takes as its argument the name of a model entity and returns its *alias*, a literal value described in Section A.5. The functions `round`(x, n) and `trunc`(x, n) convert x to a decimal string and round or truncate it to n places past the decimal point (or to $-n$ places before the decimal point if $n < 0$); similarly, `precision`(x, n) rounds x to n significant decimal digits. For `round` and `trunc`, a missing n is taken as 0, thus providing the usual rounding or truncation to an integer.

Several built-in random number generation functions are available, as listed in Table A-3. All are based on a uniform random number generator with a very long period. An initial seed n can be specified with the `-s`n command-line argument (A.23) or option `randseed`, while `-s` or

`Beta(`*a*`,` *b*`)`	*density*$(x) = x^{a-1}(1-x)^{b-1}/(\Gamma(a)\Gamma(b)/\Gamma(a+b)),\ x$ in $[0,\ 1]$
`Cauchy()`	*density*$(x) = 1/(\pi(1+x^2))$
`Exponential()`	*density*$(x) = e^{-x},\ x>0$
`Gamma(`*a*`)`	*density*$(x) = x^{a-1}e^{-x}\ /\ \Gamma(a),\ \ x\geq 0,\ a>0$
`Irand224()`	integer uniform on $[0, 2^{24})$
`Normal(`μ`,` σ`)`	normal distribution with mean μ, variance σ
`Normal01()`	normal distribution with mean 0, variance 1
`Poisson(`μ`)`	*probability*$(k) = e^{-\mu}\mu^k/k!,\ k = 0,\ 1,\ ...$
`Uniform(`*m*`,` *n*`)`	uniform on [*m*, *n*)
`Uniform01()`	uniform on [0, 1)

Table A-3: Built-in random number generation functions.

`option randseed ''` instructs AMPL to choose and print a seed. Giving no `-s` argument is the same as specifying `-s1`.

`Irand224()` returns an integer in the range $[0,\ 2^{24})$. Given the same seed, an expression of the form `floor(`*m*`*Irand224()/`*n*`)` will yield the same value on most computers when *m* and *n* are integer expressions of reasonable magnitude, i.e., $|n| < 2^{k-24}$ and $|m| < 2^k$, for machines that correctly compute *k*-bit floating-point integer products and quotients; $k \geq 47$ for most machines of current interest.

Functions that operate on sets are described in Section A.6.

A.4.2 Strings and regular expressions

In almost all contexts in a model or command where a literal string could be used, it is also possible to use a string expression, enclosed in parentheses. Strings are created by concatenation and from the built-in string and regular expression functions listed in Table A-4.

The string concatenation operator `&` concatenates its arguments into a single string; it has precedence below all arithmetic operators. Numeric operands are converted to full-precision decimal strings as though by `printf` format `%.g` (A.16).

s `&` *t*	concatenate strings *s* and *t*
`num(`*s*`)`	convert string *s* to number; error if stripping leading and trailing white space does not yield a valid decimal number
`num0(`*s*`)`	strip leading white space, and interpret as much as possible of *s* as a number, but do not raise error
`ichar(`*s*`)`	Unicode value of the first character in string *s*
`char(`*n*`)`	string representation of character *n*; inverse of `ichar`
`length(`*s*`)`	length of string *s*
`substr(`*s*`,` *m*`,` *n*`)`	*n* character substring of *s* starting at position *m*; if *n* omitted, rest of string
`sprintf(`*f*`,` *exprlist*$_{opt}$`)`	format arguments according to format string *fmt*
`match(`*s*`,` *re*`)`	starting position of regular expression *re* in *s*, or 0 if not found
`sub(`*s*`,` *re*`,` *repl*`)`	substitute *repl* for the first occurrence of regular expression *re* in *s*
`gsub(`*s*`,` *re*`,` *repl*`)`	substitute *repl* for all occurrences of regular expression *re* in *s*

Table A-4: Built-in string and regular expression functions.

There is no implicit conversion of strings to numbers, but `num(`*s*`)` and `num0(`*s*`)` perform explicit conversions. Both ignore leading and trailing white space; `num` complains if what remains is not a valid number, whereas `num0` converts as much as it can, returning 0 if it sees no numeric prefix.

The `match`, `sub`, and `gsub` functions accept strings representing regular expressions as their second arguments. AMPL regular expressions are similar to standard Unix regular expressions. Aside from certain *metacharacters,* any literal character *c* is a regular expression that matches an occurrence of itself in the target string. The metacharacter ''`.`'' is a regular expression that matches any character. A list of characters enclosed in brackets is a regular expression that matches any character in the list, and lists may be abbreviated: `[a-z0-9]` matches any lower case letter or digit. A list of characters that begins with the character `^` and is enclosed in brackets is a regular expression that matches any character *not* in the list: `[^0-9]` matches any non-digit. If *r* is a regular expression, then *r*`*` matches 0 or more occurrences of *r*, *r*`+` matches 1 or more occurrences, and *r*`?` matches 0 or 1 occurrence. `^`*r* matches *r* only if *r* occurs at the beginning of the string, and *r*`$` matches *r* only at the end of the string. Parentheses are used for grouping and `|` means ''or''; r_1 `|` r_2 matches r_1 or r_2. The special meaning of a metacharacter can be turned off by preceding it by a backslash.

In the replacement pattern (third argument) for `sub` and `gsub`, `&` stands for the whole matched string, as does `\0`, while `\1`, `\2`, ..., `\9` stand for the string matched by the first, second, ..., ninth parenthesized expression in the pattern.

Options (A.14.1) are named string values, some of which affect various AMPL commands (A.14). The current value of option *opname* is denoted `$`*opname*.

A.4.3 Piecewise-linear terms

In variable, constraint and objective declarations, piecewise-linear terms have one of the following forms:

```
<< breakpoints ; slopes >> var
<< breakpoints ; slopes >> (cexpr)
<< breakpoints ; slopes >> (var, cexpr)
<< breakpoints ; slopes >> (cexpr, cexpr)
```

where *breakpoints* is a list of breakpoints and *slopes* a list of slopes. Each such list is a comma-separated list of *cexpr*'s, each optionally preceded by an indexing expression (whose scope covers just the *cexpr*). The indexing expression must specify a set that is manifestly ordered (see A.6.2), or it can be of the special form

```
{if lexpr}
```

which causes the *expr* to be omitted if the *lexpr* is false. In commands, the more general forms

```
<< breakpoints ; slopes >> (expr)
<< breakpoints ; slopes >> (expr, expr)
```

are also allowed, and variables may appear in expressions in the *breakpoints* and *slopes*.

After the lists of slopes and breakpoints are extended (by indexing over any indexing expressions), the number of slopes must be one more than the number of breakpoints, and the breakpoints must be in non-decreasing numeric order. (There is no requirement on the order of the slopes.) AMPL interprets the result as the piecewise-linear function $f(x)$ defined as follows. Let s_j, $1 \le j \le n$, and b_i, $1 \le i \le n-1$, denote the slopes and breakpoints, respectively, and let

$b_0 = -\infty$ and $b_n = +\infty$. Then $f(0)=0$, and for $b_{i-1} \le x \le b_i$, f has slope s_i, i.e., $f'(x) = s_i$. For the forms having just one argument (either a variable *var* or a constant expression *expr*), the result is $f(var)$ or $f(expr)$. The form with two operands is interpreted as $f(var) - f(expr)$. This adds a constant that makes the result vanish when the *var* equals the *expr*.

When piecewise-linear terms appear in an otherwise linear constraint or objective, AMPL collects two or more piecewise-linear terms involving the same variable into a single term.

A.5 Declarations of model entities

Declarations of model entities have the following common form:

entity name $alias_{opt}$ $indexing_{opt}$ $body_{opt}$;

where *name* is an alphanumeric name that has not previously been assigned to an entity by a declaration, *alias* is an optional literal, *indexing* is an optional indexing expression, and *entity* is one of the keywords

```
set
param
var
arc
minimize
maximize
subject to
node
```

In addition, several other constructs are technically entity declarations, but are described later; these include `environ`, `problem`, `suffix` and `table`.

The *entity* may be omitted, in which case `subject to` is assumed. The *body* of various declarations consists of other, mostly optional, phrases that follow the initial part. Each declaration ends with a semicolon.

Declarations may appear in any order, except that each name must be declared before it is used.

As with piecewise-linear terms, a special form of indexing expression is allowed for variable, constraint, and objective declarations:

`{if` *lexpr*`}`

If the logical expression *lexpr* is true, then a simple (unsubscripted) entity results; otherwise the entity is excluded from the model, and subsequent attempts to reference it cause an error message. For example, this declaration includes the variable `Test` in the model if the parameter `testing` has been given a value greater than 100:

```
param testing;
var Test {if testing > 100} >= 0;
```

A.6 Set declarations

A set declaration has the form

set declaration:
    `set` *name* *alias*$_{opt}$ *indexing*$_{opt}$ *attributes*$_{opt}$ `;`

in which *attributes* is a list of attributes optionally separated by commas:

attribute:
    `dimen` *n*
    `within` *sexpr*
    `=` *sexpr*
    `default` *sexpr*

The dimension of the set is either the constant positive integer *n*, or the dimension of *sexpr*, or 1 by default. The phrase `within` *sexpr* requires the set being declared to be a subset of *sexpr*. Several `within` phrases may be given. The = phrase specifies a value for the set; it implies that the set will not be given a value later in a data section (A.12.1) or a command such as `let` (A.18.9). The `default` phrase specifies a default value for the set, to be used if no value is given in a data section. The = and `default` phrases are mutually exclusive. If neither is given and the set is not defined by a data statement, references to the set during model generation cause an error message. For historical reasons, := is currently a synonym for = in declarations of sets and parameters, but this use of := is deprecated.

The *sexpr* in a = or `default` phrase can be `{}`, the empty set, which then has the dimension implied by any `dimen` or `within` phrases in the declaration, or 1 if none is present. In other contexts, `{}` denotes the empty set.

Recursive definitions of indexed sets are allowed, so long as the assigned values can be computed in a sequence that only references previously computed values. For example,

```
set nodes;
set arcs within nodes cross nodes;

param max_iter = card(nodes)-1; # card(s) = number of elements in s

set step {s in 1..max_iter} dimen 2 =
    if s == 1
        then arcs
        else step[s-1] union setof {k in nodes,
               (i,k) in step[s-1], (k,j) in step[s-1]} (i,j);

set reach = step[max_iter];
```

computes in set `reach` the transitive closure of the graph represented by `nodes` and `arcs`.

A.6.1 Cardinality and arity functions

The function `card` operates on any finite set: `card(`*sexpr*`)` returns the number of members in *sexpr*. If *sexpr* is an indexing expression, the parentheses may be omitted. For example,

```
card({i in A: x[i] >= 4})
```

may also be written

```
card {i in A: x[i] >= 4}
```

The function `arity` returns the arity of its set argument; the function `indexarity` returns the arity of its argument's indexing set.

A.6.2 Ordered sets

A named one-dimensional set may have an order associated with it. Declarations for ordered sets include one of the phrases

```
ordered [ by [ reversed ] sexpr ]
circular [ by [ reversed ] sexpr ]
```

The keyword `circular` indicates that the set is ordered and wraps around, i.e., its first member is the successor of its last, and its last member is the predecessor of its first.

Sets of dimension two or higher may not currently be `ordered` or `circular`.

If set `S` is `ordered by T` or `circular by T`, then set `T` must itself be an ordered set that contains `S`, and `S` inherits the order of `T`. If the ordering phrase is `by reversed T`, then `S` still inherits its order from `T`, but in reverse order.

If `S` is `ordered` or `circular` and no ordering `by` *sexpr* is specified, then one of two cases applies. If the members of `S` are explicitly specified by a `= {`*member-list*`}` expression in the model or by a list of members in a data section, `S` adopts the ordering of this list. If `S` is otherwise computed from an assigned or default *sexpr* in the model, AMPL will retain the order of manifestly ordered sets (explained below) and is otherwise free to pick an arbitrary order.

Functions of ordered sets are summarized in Table A-5.

If S is an expression for an ordered set of n members, e is the jth member of S, and k is an integer expression, then `next(`e`,`S`,`k`)` is member $j + k$ of S if $1 \le j + k \le n$, and an error otherwise. If S is `circular`, then `next(`e`,`S`,`k`)` is member $1 + ((j + k - 1) \bmod n)$ of S. The function `nextw` (`next` with wraparound) is the same as `next`, except that it treats all ordered sets as `circular`; `prev(`e`,`S`,`k`)` ≡ `next(`e`,`S`,`$-k$`)`, and `prevw(`e`,`S`,`k`)` ≡ `nextw(`e`,`S`,`$-k$`)`.

Several abbreviations are allowed. If k is not given, it is taken as 1. If both k and S are omitted, then e must be a dummy index in the scope of an indexing expression that runs over S, for example as in `{`e `in` S`}`.

Five other functions apply to ordered sets: `first(`S`)` returns the first member of S, `last(`S`)` the last, `member(`j`,`S`)` returns the jth member of S, `ord(`e`,`S`)` and `ord0(`e`,`S`)` the ordinal position of member e in S. If e is not in S, then `ord0` returns 0, whereas `ord` complains of an error. Again, `ord(`e`) = ord(`e`,`S`)` and `ord0(`e`) = ord0(`e`,`S`)` if e is a dummy index in the scope of an indexing expression for S.

Some sets are manifestly ordered, such as arithmetic progressions, intervals, subsets of ordered sets, `if` expressions whose `then` and `else` clauses are both ordered sets, and set differences (but not symmetric differences) whose left operands are ordered.

A.6.3 Intervals and other infinite sets

For convenience in specifying ordered sets and prototypes for imported functions (A.22), there are several kinds of infinite sets. Iterating over infinite sets is forbidden, but one can check membership in them and use them to specify set orderings. The most natural infinite set is the interval, which may be closed, open, or half-open and which follows standard mathematical notation. There are intervals of real (floating-point) numbers and of integers, introduced by the keywords `interval` and `integer` respectively:

`next(`*e*`,`*S*`,`*k*`)`	member *k* positions after member *e* in set *S*
`next(`*e*`,`*S*`)`	same, with *k* = 1
`next(`*e*`)`	next member of set for which *e* is dummy index
`nextw(`*e*`,`*S*`,`*k*`)`	member *k* positions after member *e* in set *S*, wrapping around
`nextw(`*e*`,`*S*`)`	wrapping version of `next(`*e* `,` *S*`)`
`nextw(`*e*`)`	wrapping version of `next(`*e*`)`
`prev(`*e*`,`*S*`,`*k*`)`	member *k* positions before member *e* in set *S*
`prev(`*e*`,`*S*`)`	same, with *k* = 1
`prev(`*e*`)`	previous member of set for which *e* is dummy index
`prevw(`*e*`,`*S*`,`*k*`)`	member *k* positions before member *e* in set *S*, wrapping around
`prevw(`*e*`,`*S*`)`	wrapping version of `prev(`*e* `,` *S*`)`
`prevw(`*e*`)`	wrapping version of `prev(`*e*`)`
`first(`*S*`)`	first member of *S*
`last(`*S*`)`	last member of *S*
`member(`*j*`,`*S*`)`	*j*th member of *S*; $1 \le j \le$ `card(`*S*`)`, *j* integer
`ord(`*e*`,`*S*`)`	ordinal position of member *e* in *S*
`ord(`*e*`)`	ordinal position of member *e* in set for which it is dummy index
`ord0(`*e*`,`*S*`)`	ordinal position of member *e* in *S*; 0 if not present
`ord0(`*e*`)`	same as `ord(`*e*`)`
`card(`*S*`)`	number of members in set *S*
`arity(`*S*`)`	arity of *S* if *S* is a set, else 0; for use with `_SETS`
`indexarity(`*E*`)`	arity of entity *E*'s indexing set
	`card`, `arity`, and `indexarity` also apply to unordered sets

Table A-5: Functions of ordered sets.

interval:
`interval [a, b]` ≡ {*x*: `a` ≤ *x* ≤ `b`},
`interval (a, b]` ≡ {*x*: `a` < *x* ≤ `b`},
`interval [a, b)` ≡ {*x*: `a` ≤ *x* < `b`},
`interval (a, b)` ≡ {*x*: `a` < *x* < `b`},
`integer [a, b]` ≡ {*x*: `a` ≤ *x* ≤ `b` and *x* ∈ *I*},
`integer (a, b]` ≡ {*x*: `a` < *x* ≤ `b` and *x* ∈ *I*},
`integer [a, b)` ≡ {*x*: `a` ≤ *x* < `b` and *x* ∈ *I*},
`integer (a, b)` ≡ {*x*: `a` < *x* < `b` and *x* ∈ *I*}

where `a` and `b` denote arbitrary arithmetic expressions, and *I* denotes the set of integers. In function prototypes (A.22) and the declaration phrases

`in` *interval*
`within` *interval*
`ordered by` *[* `reversed` *]* *interval*
`circular by` *[* `reversed` *]* *interval*

the keyword `interval` may be omitted.

The predefined infinite sets `Reals` and `Integers` are the sets of all floating-point numbers and integers, respectively, in numeric order. The predefined infinite sets `ASCII`, `EBCDIC`, and `Display` all represent the universal set of strings and numbers from which members of any one-dimensional set are drawn; `ASCII` is ordered by the ASCII collating sequence, `EBCDIC` by the

EBCDIC collating sequence, and `Display` by the ordering used in the `display` command (Section A.16). Numbers precede literals in `Display` and are ordered numerically; literals are sorted by the ASCII collating sequence.

A.7 Parameter declarations

Parameter declarations have a list of optional attributes, optionally separated by commas:

parameter declaration:
`param` *name* *alias*$_{opt}$ *indexing*$_{opt}$ *attributes*$_{opt}$ `;`

The *attributes* may be any of the following:

attribute:
`binary`
`integer`
`symbolic`
relop expr
`in` *sexpr*
`=` *expr*
`default` *expr*

relop:
`<` `<=` `=` `==` `!=` `<>` `>` `>=`

The keyword `integer` restricts the parameter to be an integer; `binary` restricts it to 0 or 1. If `symbolic` is specified, then the parameter may assume any literal or numeric value (rounded as for set membership), and the attributes involving <, <=, >= and > are disallowed; otherwise the parameter is numeric and can only assume a numeric value.

The attributes involving comparison operators specify that the parameter must obey the given relation. The `in` attribute specifies a check that the parameter lies in the given set.

The = and `default` attributes are analogous to the corresponding ones in set declarations, and are mutually exclusive.

Recursive definitions of indexed parameters are allowed, so long as the assigned values can be computed in a sequence that only references previously computed values. For example,

```
param comb 'n choose k' {n in 0..N, k in 0..n}
   = if k = 0 or k = n then 1 else comb[n-1,k-1] + comb[n-1,k];
```

computes the number of ways of choosing *n* things *k* at a time.

In a recursive definition of a symbolic parameter, the keyword `symbolic` must precede all references to the parameter.

A.7.1 Check statements

Check statements supply assertions to help verify that correct data have been read or generated; they have the syntax

`check` `[` *indexing*$_{opt}$ `:` `]` *lexpr* `;`

Each `check` statement is evaluated when one of the commands `solve`, `write`, `solution`, or `check` is executed.

A.7.2 Infinity

`Infinity` is a predefined parameter; it is the threshold above which upper bounds are considered absent (i.e., infinite), and `-Infinity` is the threshold below which lower bounds are considered absent. Thus given

```
set A;
param Ub{A} default  Infinity;
param Lb{A} default -Infinity;
var V {i in A} >= Lb[i], <= Ub[i];
```

components of `V` for which no `Lb` value is given in a data section are unbounded below and components for which no `Ub` value is given are unbounded above. One can similarly arrange for optional lower and upper constraint bounds. On computers with IEEE arithmetic (most modern systems) `Infinity` is the IEEE ∞ value.

A.8 Variable declarations

Variable declarations begin with the keyword `var`:

variable declaration:
`var` *name* *alias*$_{opt}$ *indexing*$_{opt}$ *attributes*$_{opt}$ `;`

Optional attributes of variable declarations may be separated by commas; these attributes include

attribute:
`binary`
`integer`
`symbolic`
`>=` *expr*
`<=` *expr*
`:=` *expr*
`default` *expr*
`=` *expr*
`coeff` *indexing*$_{opt}$ *constraint* *expr*
`cover` *indexing*$_{opt}$ *constraint*
`obj` *indexing*$_{opt}$ *objective* *expr*
`in` *sexpr*
`suffix` *sufname* *expr*

As with parameters, `integer` restricts the variable to integer values, and `binary` restricts it to 0 or 1. The `>=` and `<=` phrases specify bounds, and the `:=` phrase an initial value. The `default` phrase specifies a default for initial values that may be provided in a data section (A.12.2); `default` and `:=` are mutually exclusive. The `=` *expr* phrase is allowed only if none of the previous attributes appears; it makes the variable a *defined variable* (A.8.1). Each `suffix` *sufname expr* phrase specifies an initial value for a previously declared suffix *sufname*.

If `symbolic` is specified, `in` *sexpr* must also appear and attributes requiring a numeric value, such as `>=` *expr*, are excluded. If `in` *sexpr* appears without `symbolic`, the set expression *sexpr* must be the union of intervals and discrete sets of numbers. Either way, `in` *sexpr* restricts the variable to lie in *sexpr*.

The `coeff` and `obj` phrases are for columnwise coefficient generation; they specify coefficients to be placed in the named constraint or objective, which must have been previously declared

using the placeholder `to_come` (see A.9 and A.10). The scope of *indexing* is limited to the phrase, and may have the special form

`{if` *lexpr*`}`

which contributes the coefficient only if the *lexpr* is true. A `cover` phrase is equivalent to a `coeff` phrase in which the *expr* is 1.

Arcs are special network variables, declared with the keyword `arc` instead of `var`. They may contribute coefficients to `node` constraints (A.9) via optional attribute phrases of the form

`from` *indexing*$_{opt}$ *node expr*$_{opt}$
`to` *indexing*$_{opt}$ *node expr*$_{opt}$

These phrases are analogous in syntax to the `coeff` phrase, except that the final *expr* is optional; omitting it is the same as specifying 1.

A.8.1 Defined variables

In some nonlinear models, it is convenient to define named values that contribute somehow to the constraints or objectives. For example,

```
set A;
var v {A};
var w {A};
subject to C {i in A}: w[i] = v expr;
```

where v*expr* is an expression involving the variables `v`.

As things stand, the members of `C` are constraints, and we have turned an unconstrained problem into a constrained one, which may not be a good idea. Setting option `substout` to 1 instructs AMPL to eliminate the collection of constraints `C`. AMPL does so by telling solvers that the constraints define the variables on their left-hand sides, so that, in effect, these defined variables become named common subexpressions.

When option `substout` is 1, a constraint such as `C` that provides a definition for a defined variable is called a *defining constraint*. AMPL decides which variables are defined variables by scanning the constraints once, in order of their appearance in the model. A variable is eligible to become a defined variable only if its declaration imposes no constraints on it, such as integrality or bounds. Once a variable has appeared on the right-hand side of a defining constraint, it is no longer eligible to be a defined variable — without this restriction, AMPL might have to solve implicit equations. Once a variable has been recognized as a defined variable, its subsequent appearance on the left-hand side of what would otherwise be a defining constraint results in the constraint being treated as an ordinary constraint.

Some solvers give special treatment to linear variables because their higher derivatives vanish. For such solvers, it may be helpful to treat linear defined variables specially. Otherwise, variables involved in the right-hand side of the equation for a defined variable appear to solvers as nonlinear variables, even if they are used only linearly in the right-hand side. By doing Gaussian elimination rather than conveying linear variable definitions explicitly, AMPL can arrange for solvers to see such right-hand variables as linear variables. This often causes fill-in, i.e., makes the problem less sparse, but it may give the solvers a more accurate view of the problem. When option `linelim` has its default value 1, AMPL treats linear defined variables in this special way; when option `linelim` is 0, AMPL treats all defined variables alike.

A variable declaration may have a phrase of the form = *expr*, where *expr* is an expression involving variables. Such a phrase indicates that the variable is to be defined with the value *expr*. Such defining declarations allow some models to be written more compactly.

Recognizing defined variables is not always a good idea — it leads to a problem in fewer variables, but one that may be more nonlinear and that may be more expensive to solve because of loss of sparsity. By using defining constraints (instead of using defining variable declarations) and translating and solving a problem both with `$substout = 0` and with `$substout = 1`, one can see whether recognizing defined variables is worthwhile. On the other hand, if recognizing a defined variable seems clearly worthwhile, defining it in its declaration is often more convenient than providing a separate defining constraint; in particular, if all defined variables are defined in their declarations, one need not worry about `$substout`.

One restriction on defining declarations is that subscripted variables must be defined before they are used.

A.9 Constraint declarations

The form of a constraint declaration is

constraint declaration:
[`subject to` *]* *name* *alias*$_{opt}$ *indexing*$_{opt}$
[`:=` *initial_dual* *]* *[* `default` *initial_dual* *]*
[`:` *constraint expression* *]* *[* *suffix-initializations* *]* `;`

The keyword `subject to` in a constraint declaration may be omitted but is usually best retained for clarity. The optional `:=` *initial_dual* specifies an initial guess for the dual variable (Lagrange multiplier) associated with the constraint. Again, `default` and `:=` clauses are mutually exclusive, and `default` is for initial values not given in a data section. Constraint declarations must specify a constraint in one of the following forms:

constraint expression:
expr `<=` *expr*
expr `=` *expr*
expr `>=` *expr*
cexpr `<=` *expr* `<=` *cexpr*
cexpr `>=` *expr* `>=` *cexpr*

To enable columnwise coefficient generation for the constraint, one of the *expr*s may have one of the following forms:

`to_come` + *expr*
expr + `to_come`
`to_come`

Terms for this constraint that are specified in a `var` declaration (A.8) are placed at the location of `to_come`.

Nodes are special constraints that may send flow to or receive flow from arcs. Their declarations begin with the keyword `node` instead of `subject to`. Pure transshipment nodes do not have a constraint body; they must have ‘‘flow in’’ equal to ‘‘flow out’’. Other nodes are sources or sinks; they specify constraints in one of the forms above, except that they may not mention `to_come`, and exactly one *expr* must have one of the following forms:

```
net_in + expr
net_out + expr
expr + net_in
expr + net_out
net_in
net_out
```

The keyword `net_in` is replaced by an expression representing the net flow into the node: the terms contributed to the node constraint by `to` phrases of `arc` declarations (A.8), minus the terms contributed by `from` phrases. The treatment of `net_out` is analogous; it is the negative of `net_in`.

The optional *suffix-initialization* phrases each have the form

suffix-initialization:
    `suffix` *sufname expr*

optionally preceded by a comma, where *sufname* is a previously declared suffix.

A.9.1 Complementarity constraints

For expressing complementarity constraints, in addition to the forms above, constraint declarations may have the form

name alias$_{opt}$ *indexing*$_{opt}$ `:` *constr*$_1$ `complements` *constr*$_2$ `;`

in which *constr*$_1$ and *constr*$_2$ consist of 1, 2, or 3 expressions separated by the operators <=, >= or =. In *constr*$_1$ and *constr*$_2$ together, there must be a total of two explicit inequality operators, with = counting as two. A complementarity constraint is satisfied if both *constr*$_1$ and *constr*$_2$ hold and at least one inequality is tight, i.e., satisfied as an equality. If one of *constr*$_1$ or *constr*$_2$ involves two inequalities, then the constraint must have one of the forms

expr$_1$ `<=` *expr*$_2$ `<=` *expr*$_3$ `complements` *expr*$_4$
expr$_3$ `>=` *expr*$_2$ `>=` *expr*$_1$ `complements` *expr*$_4$
expr$_4$ `complements` *expr*$_1$ `<=` *expr*$_2$ `<=` *expr*$_3$
expr$_4$ `complements` *expr*$_3$ `>=` *expr*$_2$ `>=` *expr*$_1$

In all of these cases, the constraint requires the inequalities to hold, with

expr$_4$ $\geq$ 0 `if` *expr*$_1$ `=` *expr*$_2$
expr$_4$ $\leq$ 0 `if` *expr*$_2$ `=` *expr*$_3$
expr$_4$ `=` 0 `if` *expr*$_1$ `<` *expr*$_2$ `<` *expr*$_3$

For expressing mathematical programs with equilibrium constraints, complementarity constraints may coexist with other constraints and objectives.

Solvers see complementarity constraints in the standard form

expr `complements` *lower bound* `<=` *variable* `<=` *upper bound*

A synonym (A.19.4), `_scvar{i in 1.._sncons}`, indicates which variable, if any, complements each constraint the solver sees. If `_scvar[i] in 1.._snvars`, then variable `_svar[_scvar[i]]` complements constraint `_scon[i]`; otherwise `_scvar[i] == 0`, and `_con[i]` is an ordinary constraint. The synonyms `_cconname{1.._nccons}` are the names of the complementarity constraints as the modeler sees them.

A.10 Objective declarations

The declaration of an objective is one of

objective declaration:
`maximize` *name* *alias*$_{opt}$ *indexing*$_{opt}$ [`:` *expression*] [*suffix-initializations*] `;`
`minimize` *name* *alias*$_{opt}$ *indexing*$_{opt}$ [`:` *expression*] [*suffix-initializations*] `;`

and may specify an expression in one of the following forms:

expression:
expr
`to_come` + *expr*
expr + `to_come`
`to_come`

The `to_come` forms permit columnwise coefficient generation, as for constraints (A.9). Specifying none of the above expressions is the same as specifying ''`: to_come`''. *Suffix-initializations* may appear as in constraint declarations.

If there are multiple objectives, the one sent to a solver can be set by the `objective` command; see section A.18.6. By default, all objectives are sent.

A.11 Suffix notation for auxiliary values

Variables, constraints, objectives and problems have a variety of associated auxiliary values. For example, variables have bounds and reduced costs, and constraints have dual values and slacks. Such values are accessed as *name* `.` *suffix*, where *name* is a simple or subscripted variable, constraint, objective or problem name, and `.` *suffix* is one of the possibilities listed in Tables A-6, A-7, and A-8.

For a constraint, the `.body`, `.lb`, and `.ub` values correspond to a modified form of the constraint. If the constraint involves a single inequality, subtract the right-hand side from the left, then move any constant terms to the right-hand side; if the constraint involves a double inequality, similarly subtract any constant terms in the middle from all three expressions (left, middle, right). Then the constraint has the form $lb \le body \le ub$, where *lb* and *ub* are (possibly infinite) constants.

The following rules determine lower and upper dual values (*c* `.ldual` and *c* `.udual`) for a constraint *c*. The solver returns a single dual value, *c* `.dual`, which might apply either to $body \ge lb$ or to $body \le ub$. For an equality constraint ($lb = ub$), AMPL uses the sign of *c* `.dual` to decide. For a minimization problem, *c* `.dual` > 0 implies that the same optimum would be found if the constraint were $body \ge lb$, so AMPL sets *c* `.ldual` = *c* `.dual` and *c* `.udual` = 0; similarly, *c* `.dual` < 0 implies that *c* `.ldual` = 0 and *c* `.udual` = *c* `.dual`. For a maximization problem, the inequalities are reversed.

For inequality constraints ($lb < ub$), AMPL uses nearness to bound to decide whether *c* `.ldual` or *c* `.udual` equals *c* `.dual`. If $body - lb < ub - body$, then *c* `.ldual` = *c* `.dual` and *c* `.udual` = 0; otherwise, *c* `.ldual` = 0 and *c* `.udual` = *c* `.dual`.

Model declarations may reference any of the suffixed values described in Tables A-6, A-7 and A-8. This is most often useful in new declarations that are introduced after one model has already been translated and solved. In particular, the suffixes `.val` and `.dual` are provided so that new constraints can refer to current optimal values of the primal and dual variables and of the objective.

`.astatus`	AMPL status (A.11.2)
`.init`	current initial guess
`.init0`	initial initial guess (set by `:=`, data, or `default`)
`.lb`	current lower bound
`.lb0`	initial lower bound
`.lb1`	weaker lower bound from presolve
`.lb2`	stronger lower bound from presolve
`.lrc`	lower reduced cost (for `var >= lb`)
`.lslack`	lower slack (`val - lb`)
`.rc`	reduced cost
`.relax`	ignore integrality restriction if positive
`.slack`	min(`lslack`, `uslack`)
`.sstatus`	solver status (A.11.2)
`.status`	status (A.11.2)
`.ub`	current upper bound
`.ub0`	initial upper bound
`.ub1`	weaker upper bound from presolve
`.ub2`	stronger upper bound from presolve
`.urc`	upper reduced cost (for `var <= ub`)
`.uslack`	upper slack (`ub - val`)
`.val`	current value of variable

Table A-6: Dot suffixes for variables.

For a complementarity constraint, suffix notations like *constraint* `.lb`, *constraint* `.body`, etc., are extended so that *constraint* `.L`*suffix* and *constraint* `.R`*suffix* correspond to $constr_1$ `.` *suffix* and $constr_2$ `.` *suffix*, respectively, and *complementarity-constraint* `.slack` (or the unadorned name) stands for a measure of the extent to which the complementarity constraint is satisfied: if $constr_1$ and $constr_2$ each involve one inequality, the new measure is

$\texttt{min}(constr_1\texttt{.slack},\ constr_2\texttt{.slack}),$

which is positive if both are satisfied as strict inequalities, 0 if the complementarity constraint is satisfied exactly, and negative if at least one of $constr_1$ or $constr_2$ is violated. For constraints of the form $expr_1$ <= $expr_2$ <= $expr_3$ `complements` $expr_4$, the `.slack` value is

$\texttt{min}(expr_2\texttt{-}expr_1,\ \ expr_4)\ \texttt{if}\ expr_1\ \texttt{>=}\ expr_2$
$\texttt{min}(expr_3\texttt{-}expr_2,\ \texttt{-}expr_4)\ \texttt{if}\ expr_3\ \texttt{<=}\ expr_2$
$\texttt{-abs(}expr_4\texttt{)}\ \texttt{if}\ expr_1\ \texttt{<}\ expr_2\ \texttt{<}\ expr_3$

so in all cases, the `.slack` value is 0 if the complementarity constraint holds exactly and is negative if one of the requisite inequalities is violated.

A.11.1 Suffix declarations

Suffix declarations introduce new suffixes, which may be assigned values in subsequent declarations, `let` commands and function invocations (with `OUT` arguments, A.22). Suffix declarations begin with the keyword `suffix`:

`.astatus`	AMPL status (A.11.2)
`.body`	current value of constraint *body*
`.dinit`	current initial guess for dual variable
`.dinit0`	initial initial guess for dual variable (set by `:=`, data, or `default`)
`.dual`	current dual variable
`.lb`	lower bound
`.lbs`	`lb` for solver (adjusted for fixed variables)
`.ldual`	lower dual value (for `body >= lb`)
`.lslack`	lower slack (`body - lb`)
`.slack`	min(`lslack`, `uslack`)
`.sstatus`	solver status (A.11.2)
`.status`	status (A.11.2)
`.ub`	upper bound
`.ubs`	`ub` for solver (adjusted for fixed variables)
`.udual`	upper dual value (for `body <= ub`)
`.uslack`	upper slack (`ub - body`)

Table A-7: Dot suffixes for constraints.

`.val`	current value of objective

Table A-8: Dot suffix for objectives.

suffix declaration:
`suffix` *name* *alias*$_{opt}$ *attributes*$_{opt}$ `;`

Optional attributes of suffix declarations may be separated by commas; these attributes include

attribute:
`binary`
`integer`
`symbolic`
`>=` *expr*
`<=` *expr*
direction

direction:
`IN`
`OUT`
`INOUT`
`LOCAL`

At most one *direction* may be specified; AMPL assumes `INOUT` if no *direction* is given. These directions are from a solver's perspective: `IN` suffix values are input to the solver; `OUT` suffix values are assigned by the solver; `INOUT` values are both `IN` and `OUT`; and `LOCAL` values are not seen by the solver.

Symbolic suffixes are declared with the `symbolic` attribute; appending `_num` to the name of a symbolic suffix gives the name of an associated numeric suffix; solvers see the associated numeric value. If *symsuf* is a symbolic suffix, option *symsuf* `_table` connects *symsuf* with *symsuf* `_num` as follows. Each line of `$`*symsuf* `_table` should begin with a numeric limit value,

followed by a string value and optional comments, all separated by white space. The numeric limit values must increase with each line. The string value with the greatest numeric limit value less than or equal to the .*sufname*_num value is the associated string value. If the .*sufname*_num value is less than the limit value in the first line of $*symsuf*_table, then the .*symsuf*_num value is used as the .*symsuf* value.

A.11.2 Statuses

Some solvers maintain a *basis* and distinguish among *basic* and various kinds of *nonbasic* variables and constraints. AMPL's built-in symbolic suffix .sstatus permits solvers to return basis information to AMPL and, in a subsequent solve (A.18.1), to be given the previously optimal basis, which sometimes leads to solving the updated problem faster.

AMPL's drop/restore status (A.18.6) of constraints and its fix/unfix status (A.18.7) of variables is reflected in the built-in symbolic suffix .astatus. The built-in symbolic suffix .status is derived from .astatus and .sstatus: if the variable or constraint, say *x*, is in the current problem, *x*.status = *x*.sstatus; otherwise *x*.status = *x*.astatus. AMPL assigns *x*.astatus_num = 0 if *x* is in the current problem, so the rule for determining .status is

x.status = if *x*.astatus_num == 0 then *x*.sstatus else *x*.astatus.

When option astatus_table has its default value, *x*.astatus = 'in' when *x*.astatus_num = 0.

A.12 Standard data format

AMPL supports a standard format to describe the sets and parameter values that it combines with a model to yield a specific optimization problem.

A data section consists of a sequence of tokens (literals and strings of printing characters) separated by white space (spaces, tabs, and newlines). Tokens include keywords, literals, numbers, and the delimiters ( ) [ ] : , ; := *. A statement is a sequence of tokens terminated by a semicolon. Comments may appear as in declarations. In all cases, arrangement of data into neat rows and columns is for human readability; AMPL ignores formatting.

A data section begins with a data command and ends with end-of-input or with a command that returns to model mode (A.14).

In a data section, model entities may be assigned values in any convenient order, independent of their order of declaration.

A.12.1 Set data

Statements defining sets consist of the keyword set, the set name, an optional :=, and the members. A one-dimensional set is most easily specified by listing its members, optionally separated by commas. The single or double quotes of a literal string may be omitted if the string is alphanumeric but does not specify a number.

An *object* in a data section may be a number or a character string. As in a model, a character string may be specified by surrounding it with quotes (' or "). Since so many strings appear in data, however, AMPL allows data statements to drop the quotes around any string that consists only

of characters that may occur in a name or number, unless quotes are needed to distinguish a string from a number.

The general form of a set data statement is

set-data-statement:
    `set` *set-name* `:=` *set-spec* *set-spec* ... `;`

set-spec:
    set-template$_{opt}$ *member-list*
    set-template$_{opt}$ *member-table* *member-table* ...

The *set-name* must be the name of an individually declared set, or the subscripted name of a set from an indexed collection. The optional template has the form

set-template:
    `(` *templ-item*`,` *templ-item*`,` ... `)`

templ-item:
    object
    `*`
    `:`

where the number of *templ-item*s must equal the dimension of the named set. If no template is given, a template of all `*`'s is assumed.

There are two forms of *set-spec*, list format and table format. The list format of *set-spec* is

member-list:
    member-item *member-item* ...

member-item:
    object *object* ...

The number of *object*s in a *member-item* must match the number of `*`'s in the preceding template, which may not have `:` as a *templ-item*; the *object*s are substituted for the `*`'s, from left to right, to produce a member that is added to the set being specified. In the special case that the template contains no `*`'s, the *member-list* should be empty, while the template itself specifies one member to be added.

The table format of *set-spec* looks like this:

member-table:
    `(tr)`$_{opt}$ *t-header* *t-header* ... `:=`
    row-label ± ± ±
    row-label ± ± ±
    ...

t-header:
    `:` *object* *object* ...

row-label:
    object *object* ...

There must be at least one *t-header*, at least one *object* in each *row-label*, and as many *t-header*'s and *row-label*'s as `*`'s and `:`'s in the preceding template. If the preceding template involves any `:`'s, there must be as many `:`'s as *t-headers*; otherwise if the optional `(tr)` appears, the initial `*`'s are treated as `:`'s, and if `(tr)` does not appear, the final `*`'s are treated as `:`'s. Each table entry shown as ± must be either a `+` or a `-` symbol. Each `-` entry is ignored, while each `+` entry's *row-label*s are substituted for the template's `*`'s in sequence, and the *object*s in the *t-header*s corresponding to the `+` are substituted for the `:`'s to produce a set member.

To define a compound set, one can list all members. Each member is a parenthesized, comma-separated list of components, and successive members have an optional comma between them. Alternatively, one can describe the members of a two-dimensional set by a table or sequence of tables. In such a table, the row labels are for the first subscript, the columns for the second; ''+'' stands for a pair that occurs in the set, and ''-'' for a pair that does not. The colon introduces a table, and is mandatory in this context. If `(tr)` precedes the colon, the table is transposed, interchanging the roles of rows and columns.

In general, a `set` statement involves a sequence of 1D and 2D `set` tables. 1D tables start with either a new template (after which a `:=` is optional) or with `:=` alone, in which case the previous template is retained. The default (initial) template is `(*,...,*)`, that is, as many `*`'s as the set's dimension. 2D tables start with an optional new template, followed by `:` or `(tr)` and an optional colon, followed by a list of column labels and a `:=`. Templates containing no `*`'s stand for themselves. The effect of `(tr)` persists until a new template appears.

For indexed sets, each component set must be given in a separate data statement. It is not necessary to specify subset members in the same order as in their parent set.

A.12.2 Parameter data

There are two forms of the statement that specifies parameter data or variable initial values. The first form is analogous to the set data statement:

param-data-statement:
    `param` *param-name* *param-default*$_{opt}$ `:=` *param-spec* *param-spec* ... `;`

param-spec:
    param-template$_{opt}$ *value-list*
    param-template$_{opt}$ *value-table* *value-table* ...

with the addition of an optional *param-default* that will be described below. The *param-name* is usually the name of a parameter declared in the model, but may also be the name of a variable or constraint; the keyword `var` may be used instead of `param` to make the distinction clear.

The `param` statement's templates have the same content as in the set data statement, but are given in brackets (like subscripts) rather than parentheses:

param-template:
    `[` *templ-item*`,` *templ-item*`,` ... `]`

templ-item:
    object
    `*`
    `:`

The *value-list* is like the previously defined *member-list*, except that it also specifies a parameter or variable value:

value-list:
    value-item *value-item* ...

value-item:
    object *object* ... *entry*

The objects are substituted for `*`'s in the template to define a set member, and the parameter or variable indexed by this set member is assigned the value associated with the *entry* (see below).

The *value-table* is like the previously defined *member-table*, except that its *entry*s are values rather than + or −:

value-table:
    (tr)$_{opt}$: *t-header* *t-header* *t-header* ... :=
    row-label *entry* *entry* *entry*
    row-label *entry* *entry* *entry*
    ...

t-header:
    : *object* *object* ...

row-label:
    object *object* ...

entry:
    number
    string
    default-symbol

As in `set` statements, the notation `(tr)` means *transpose*; it implies a 2D table, and a `:` after it is optional. It remains in effect until a new template appears.

A table may be given in several chunks of columns.

Each *entry*'s *row-label* and *t-header* entries are substituted for `*`'s and `:`'s in the template to define a set member, and the parameter or variable indexed by this set member is assigned the value specified by the *entry*. The *entry* may be a number for variables and for parameters that take numerical values, or a string for variables and parameters declared with the attribute `symbolic`. An *entry* that is the default symbol (see below) is ignored.

The second form of parameter data statement provides for the definition of multiple parameters, and also optionally the set over which they are indexed:

param-data-alternate:
    `param` *param-default*$_{opt}$:
        param-name *param-name* ... := *value-item* *value-item* ... ;

    `param` *param-default*$_{opt}$: *set-name*:
        param-name *param-name* ... := *value-item* *value-item* ... ;

The named parameters must all have the same dimension. If the optional *set-name* is specified, its membership is also defined by this statement. Each *value-item* consists of an optional template followed by a list of objects and a list of values:

value-item:
    template$_{opt}$ *object* ... *entry* *entry* ...

An initial template of all `*`'s (as many as the common dimension of the named parameters) is assumed, and a template remains in effect until a new one appears. The objects must be equal in number to the number of `*`'s in the current template; when substituted for the `*`'s in the current template, they define a set member. If a set is being defined, this member is added to it. The parameters indexed by this member are assigned the values associated with the subsequent *entry*s, which obey the same rules as the table *entry*s previously described. Values are assigned in the order in which the parameters' names appeared at the beginning of the statement; the number of *entry*s must equal the number of named parameters.

A `param` data statement's optional `default` phrase has the form

param-default:
`default` *number*

If this phrase is present, any parameter named but not explicitly assigned a value in the statement is given the value of *number*.

A data item may be specified as ''`.`'' rather than an explicit value. This is normally taken as a missing value, and a reference to it in the model would cause an error message. If there is a `default` value, however, each missing value is determined from that default. A default value may be specified either through a default phrase in a parameter's declaration in the model (A.7), or from an optional phrase

`default` *r*

that follows the parameter's name in the data statement. In the latter case, *r* must be a numeric constant.

Default-value symbols may appear in both 1D and 2D tables. The *default-symbol* is initially a dot (`.`). A stack of default-value symbols is maintained, with the current symbol at the top. The `defaultsym` statement (which is recognized only in a data section) pushes a new symbol onto the stack, and `nodefaultsym` pushes a ''no symbol'' indicator onto the stack. The statement

```
defaultsym;
```

(without a symbol) pops the stack.

Parameters having three or more indices may be given values by a sequence of 1D and 2D tables, one or more for each slice of nondefault values.

In summary, a `param` statement defining one indexed parameter starts with the keyword `param` and the name of the parameter, followed by an optional default value and an optional `:=`. Then comes a sequence of 1D and 2D `param` tables, which are similar to 1D and 2D `set` tables, except that templates involve square brackets rather than parentheses, 2D tables contain numbers (or, for a symbolic parameter, literals) rather than +'s and −'s, and 1D tables corresponding to a template of k *'s contain $k + 1$ rather than k columns, the last being a column of numbers or default symbols (`.`'s). A special form, the keyword `param`, an optional default value, and a single (untransposed) 2D table, defines several parameters indexed by a common set, and another special form, `param` followed by the parameter name, an optional `:=`, and a numeric value, defines a scalar parameter.

Variable and constraint names may appear in data sections anywhere that a parameter name may appear, to specify initial values for variables and for the dual variables associated with constraints. The rules for default values are the same as for parameters. The keyword `var` is a synonym for `param` in data statements.

A.13 Database access and tables

AMPL's `table` facility permits obtaining data from and returning data to an external medium, such as a database, file, or spreadsheet. A `table` declaration establishes connections between columns of an external relational table and sets, parameters, variables and expressions in AMPL. The `read table` and `write table` commands use these connections to read data values into AMPL from tables and write them back. AMPL uses table *handlers* to implement these connections. Built-in table handlers permit reading and writing ''`.tab`'' and ''`.bit`'' files to save and restore values and experiment with AMPL's table facilities; to access databases and spreadsheets, at

least one other handler must be installed or `loaded` (A.22). The built-in set `_HANDLERS` names the currently available handlers, and the symbolic parameter `_handler_lib{_HANDLERS}` tells which shared library each handler came from.

Table declarations have the form

table-declaration:
`table` *table-name* *indexing*$_{opt}$ *in-out*$_{opt}$ *string-list*$_{opt}$ `:`
key-spec `,` *data-spec* `,` *data-spec* `,` ... `;`

in-out is one of `IN`, `OUT`, or `INOUT`; `IN` means into AMPL, `OUT` means out of AMPL, and `INOUT` means both. `INOUT` is assumed if *in-out* is not given. The optional *string-list* gives the names of drivers, files, and driver-specific options that are used to access external data; the contents depend on the handler used for the table and perhaps on the operating system.

The *key-spec* in a `table` declaration specifies key columns that uniquely identify the data to be accessed:

set-io$_{opt}$ `[`*key-col-spec* `,` *key-col-spec* `,` ...`]`

The optional *set-io* phrase has the form

set-name *arrow*

in which *arrow* is one of `<-`, `->`, or `<->`; it points in the direction that the information is moved, from the key columns to the AMPL set (by `read table`), from the set to the columns (by `write table`), or both, depending on *in-out*. Each *key-col-spec* names a column in the external table and associates a dummy index with it. A *key-col-spec* of the form

key-name

uses *key-name* for both purposes, and a *key-col-spec* of the form

key-name `~` *data-col-name*

introduces *key-name* as the dummy index and *data-col-name* as the name of the column in the external medium; *data-col-name* may be a name, quoted string, or parenthesized string expression.

Each *data-spec* names a data column in the external table. In the simplest case, the external name and the AMPL name are the same. If not, however, an external name can be associated with an internal name with the syntax

data-spec:
param-name `~` *data-col-name*

Each *data-spec* optionally ends with one of `IN`, `OUT`, or `INOUT`, which overrides the default table direction and indicates whether `read table` should read the column into AMPL (`IN` or `INOUT`), and whether `write table` should write the column to the external medium (`OUT` or `INOUT`).

Special syntax permits use of an indexing expression to describe one or more columns of data:

indexing *expr-col-desc*
indexing `(` *expr-col-desc* `,` *expr-col-desc* `,` ... `)`

in which *expr-col-desc* has the form

expr `[` `~` *colname* `]` *in-out*$_{opt}$

Another special syntax permits iterating data columns:

indexing `<` *data-spec* `,` *data-spec* `,` ... `>`

The latter may not be nested, but may contain the former.

After a table declaration, data access is done with

```
read table table-name subscript_opt ;
write table table-name subscript_opt ;
```

which refer back to the information given in the `table` declaration.

A.14 Command language overview

AMPL recognizes the commands listed in Table A-9. Commands are not part of a model, but cause AMPL to act as described below.

The command environment recognizes two modes. In *model mode*, where AMPL starts upon invocation, it recognizes model declarations (A.5) and all of the commands described below. Upon seeing a `data` statement, it switches to *data mode*, where only data-mode statements (A.12) are recognized. It returns to model mode upon encountering a keyword that cannot begin a data-mode statement, or the end of the file. Commands other than `data`, `end`, `include`, `quit` and `shell` also cause AMPL to enter model mode.

A phrase of the form

```
include filename
```

causes the indicated file to be interpolated. Here, and in subsequent contexts where a *filename* appears, if *filename* involves semicolons, quotes, white space, or non-printing characters, it must be given as a literal, i.e., '*filename*' or "*filename*". In contexts other than `include`, *filename* may also be a parenthesized string expression (A.4.2). `include` commands may be nested; they are recognized in both model and data mode. The sequences

```
model; include filename
data; include filename
```

may be abbreviated

```
model filename
data filename
```

The `commands` command is analogous to `include`, but is a statement and must be terminated by a semicolon. When a `data` or `commands` command appears in a compound command (i.e., the body of a loop or the `then` or `else` part of an `if` command (A.20.1), or simply in a sequence of commands enclosed in braces), it is executed when the flow of control reaches it, instead of when the compound command is being read. In this case, if the `data` or `commands` command does not specify a file, AMPL reads commands or data from the current input file until it encounters either an `end` command or the end of the current file.

For `include` phrases as well as `model`, `data`, and `commands` commands, files with simple names, e.g., not involving a slash (/), are sought in directories (folders) specified by option `ampl_include` (A.14.1): each nonblank line of `$ampl_include` specifies a directory; if `$ampl_include` is empty or entirely blank, files are sought in the current directory.

The option `insertprompt` (default `'<%d>'`) specifies an insertion prompt that immediately precedes the usual prompt for input from the standard input. If present, `%d` is the current insert

`call`	invoke imported function
`cd`	change current directory
`check`	perform all `check` commands
`close`	close file
`commands`	read and interpret commands from a file
`data`	switch to data mode; optionally include file contents
`delete`	delete model entities
`display`	print model entities and expressions; also `csvdisplay` and `_display`
`drop`	drop a constraint or objective
`end`	end input from current input file
`environ`	set environment for a problem instance
`exit`	exit AMPL with status value
`expand`	show expansion of model entities
`fix`	freeze a variable at its current value
`include`	include file contents
`let`	change data values
`load`	load dynamic function library
`model`	switch to model mode; optionally include file contents
`objective`	select an objective to be optimized
`option`	set or display option values
`print`	print model entities and expressions unformatted
`printf`	print model entities and expressions formatted
`problem`	define or switch to a named problem
`purge`	remove model entities
`quit`	terminate AMPL
`read`	take input from a file
`read table`	take input from a data table
`redeclare`	change declaration of entity
`reload`	reload dynamic function library
`remove`	remove file
`reset`	reset specified entities to their initial state
`restore`	undo a `drop` command
`shell`	temporary escape to operating system to run commands
`show`	show names of model entities
`solexpand`	show expansion as seen by solver
`solution`	import variable values as if from a solver
`solve`	send current instance to a solver and retrieve solution
`update`	allow updating data
`unfix`	undo a `fix` command
`unload`	unload dynamic function library
`write`	write out a problem instance
`write table`	write data to a data table
`xref`	show dependencies among entities

Table A-9: Commands.

level, i.e., nesting of `data` and `commands` commands specifying files and appearing within a compound command.

A.14.1 Options and environment variables

AMPL maintains the values of a variety of *options* that influence the behavior of commands and solvers. Options resemble the ''environment variables'' of the Windows and Unix operating systems; in fact AMPL inherits its initial options from the environments of these systems. AMPL supplies its own defaults for many options, however, if they are not inherited in this way.

The `option` command provides a way to examine and change options. It has one of the forms

```
option redirection_opt ;
option opname [ evalue ] [ , opname [ evalue ] ... ] redirection_opt ;
```

The first form prints all options that have been changed or whose default may be provided by AMPL. In the second form, if an *evalue* is present, it is assigned to *opname*; otherwise the value (a character string) currently associated with *opname* is printed. An *opname* is an option name optionally preceded by an environment name (A.18.8) and a period. The option name also may be a name-pattern, which is a name containing one or more `*`'s. In a name-pattern, a `*` stands for an arbitrary sequence, possibly empty, of name characters, and thus may match multiple names; for example

```
option *col*;
```

lists all options whose names contain the string ''`col`''. Specific environment or option names may also be given by parenthesized string expressions.

An *evalue* is a white-space-separated sequence of one or more literals, numbers, parenthesized string expressions, and references to options of the form `$`*opname* or `$$`*opname*, in which *opname* contains no `*`'s; in general, `$`*opname* means the current value of option *opname*, and `$$`*opname* means the default value, i.e., the value inherited from the operating system, if any, or provided by AMPL. The quotes around a literal may be omitted if what remains is a name or number. The displayed option values are in a format that could be read as an `option` command.

A.15 Redirection of input and output

An optional *redirection* phrase can be used with a variety of AMPL commands to capture their output in a file for subsequent processing. It applies to all forms of `display` and `print` and also to most other commands that can produce output, such as `solve`, `objective`, `fix`, `drop`, `restore`, and `expand`.

A redirection has one of the forms

```
> filename
>> filename
< filename  (for read command)
```

in which *filename* may have any of the forms that can appear in `data` and `commands` commands (A.14). The file is opened the first time a command specifies *filename* in a redirection; the first form of *redirection* causes the file to be overwritten upon first being opened, while the second form causes output to be appended to the current contents. The form `<` *filename* is used only for input from the `read` command (A.17). Once open, *filename* remains open until a `reset` or unless explicitly closed by a `close` command:

```
close filenames_opt ;
```

As long as *filename* remains open, output forms of *redirection* causes output to be appended to the file's current contents. A `close` command without a filename closes all open files and pipes. A `close` command may specify a comma-separated list of filenames. The variant

```
remove filename ;
```

closes and deletes *filename*.

A.16 Printing and display commands

The `display`, `print`, and `printf` commands print arbitrary expressions. They have the forms

```
display [ indexing: ] disparglist redirection_opt ;
print   [ indexing: ] arglist redirection_opt ;
printf  [ indexing: ] fmt , arglist redirection_opt ;
```

If *indexing* is present, its scope extends to the end of the command, and it causes one execution of the command for each member in the *indexing* set. The format string *fmt* is like a `printf` format string in the C programming language and is explained more fully below.

An *arglist* is a (possibly empty, comma-separated) list of expressions and *iterated-arglist*s; an *iterated-arglist* has one of the forms

```
indexing expr
indexing ( nonempty-arglist )
```

where *expr* is an arbitrary expression. The *expr*s can also involve simple or subscripted variable, constraint, and objective names; a constraint name represents the constraint's current dual value. A *disparglist* is described below.

The optional *redirection* (A.15) causes output to be sent to a file instead of appearing on the standard output.

The `print` command prints the items in its *arglist* on one line, separated by spaces and terminated by a newline; the separator may be changed with option `print_separator`. Literals are quoted only if they would have to be quoted in data mode. By default, numeric expressions are printed to full precision, but this can be changed with option `print_precision` or option `print_round`, as described below.

The `printf` command prints the items in its *arglist* as dictated by its format string *fmt*. It behaves like the `printf` function in C. Most characters in the format string are printed verbatim. Conversion specifications are an exception. They start with a `%` and end with a format letter, as summarized in Table A-10. Between the `%` and the format letter there may be any of `-`, for left-justification; `+`, which forces a sign; `0`, to pad with leading zeros; a minimum field width; a period; and a precision giving the maximum number of characters to be printed from a string or digits to be printed after the decimal point for `%f` and `%e` or significant digits for `%g` or minimum number of digits for `%d`. Field widths and precisions are either decimal numbers or a `*`, which is replaced by the value of the next item in the *arglist*. Each conversion specification consumes one or (when `*`'s are involved) more items from the *arglist* and formats the last item it consumes. With `%g`, a precision of 0 (`%.0g` or `%.g`) specifies the shortest decimal string that rounds to the value being formatted. The standard C escape sequences are allowed: `\a` (alert or bell), `\b` (backspace), `\f` (formfeed), `\n` (newline), `\r` (carriage return), `\t` (horizontal tab), `\v` (vertical tab), `\x`*d* and `\x`*dd*,

`%d`	signed decimal notation
`%i`	signed decimal notation (same as `%d`)
`%u`	unsigned decimal notation
`%o`	unsigned octal notation, without leading 0
`%x`	unsigned hexadecimal, using `abcdef`, without leading `0x`
`%X`	unsigned hexadecimal, using `ABCDEF`, without leading `0X`
`%c`	single character
`%s`	string
`%q`	quote string appropriately for data values
`%Q`	always quote string
`%f`	double-precision floating-point
`%e`	double-precision floating-point, exponential notation using `e`
`%E`	double-precision floating-point, exponential notation using `E`
`%g`	double-precision floating-point, using `%f` or `%e`
`%G`	double-precision floating-point, using `%f` or `%E`
`%%`	literal `%`

Table A-10: Conversion specifications in `printf` formats.

where *d* denotes a hexadecimal digit, and *d*, *dd* and *ddd*, where *d* denotes an octal digit. Format `%q` prints a string value with data-section quoting rules; format `%Q` always quotes the string.

The `sprintf` function (A.4.2) formats its argument list according to a format string that uses the same conversion specifications.

The `display` command formats various entities in tables or lists, as appropriate. Its *displarglist* is similar to an *arglist* for `print` or `printf`, except that an item to be displayed can also be a set expression or the unsubscripted name of an indexed parameter, variable, constraint, or set; furthermore iterated *arglist*s cannot be nested, i.e., they are restricted to the forms

indexing expr
indexing (*exprlist*)

where *exprlist* is a nonempty, comma-separated list of expressions. The `display` command prints scalar expressions and sets individually, and partitions indexed entities into groups having the same number of subscripts, then prints each group in its own table or sequence of tables.

By default, the `display` command rounds numeric expressions to six significant figures, but this can be changed with the options `display_precision` or `display_round`, as described below.

Several options whose names end with `_precision` control the precision with which floating-point numbers are converted to printable values; positive values imply rounding to that many significant figures, and 0 or other values imply rounding to the shortest decimal string that, when properly rounded to a machine number, would yield the number in question. If set to integral values, `$display_round` and `$print_round` override `$display_precision` and `$print_precision`, respectively, and similarly for the analogous options in Table A-11. For example, `$display_round` *n* causes the `display` command to round numeric values to *n* places past the decimal point (or to $-n$ places before the decimal point if $n < 0$). A negative precision with `%f` formats as for the `print` command with `print_round` negative. Options that affect printing include those shown in Table A-11.

csvdisplay_precision	precision for _display and csvdisplay (0 is full precision)
csvdisplay_round	rounding for _display and csvdisplay ('' is full precision)
display_1col	maximum elements for a 1D table to be displayed one element per line
display_eps	display absolute numeric values < $display_eps as zero
display_max_2d_cols	if > 0, maximum data columns in a 2D display
display_precision	precision for display command when $display_round is not numeric
display_round	places past decimal for display command to round
display_transpose	transpose tables if rows – columns < $display_transpose
display_width	maximum line length for print and display commands
expand_precision	precision for expand command when $expand_round is not numeric
expand_round	places past decimal for expand command to round
gutter_width	separation between columns for display command
objective_precision	precision for objective value displayed by solver
omit_zero_cols	if nonzero, omit all-zero columns from displays
omit_zero_rows	if nonzero, omit all-zero rows from displays
output_precision	precision used in nonlinear expression (.nl) files
print_precision	precision for print command when $print_round is not numeric
print_round	places past decimal for print command to round
print_separator	separator for values printed by print command
solution_precision	precision for solve or solution command when $solution_round is not numeric
solution_round	places past decimal for solve or solution command to round

Table A-11: Options that control printing.

Commands _display and csvdisplay are variants that emit tables in a more regular format than does display: each line of a table starts with *s* subscripts and ends with *k* items, all separated by commas. _display and csvdisplay differ in the table headers they emit. The header for _display consists of a line starting with _display and followed by three integers *s*, *k*, and *n* (the number of table lines that follow the header), each preceded by a space. If $csvdisplay_header is 1, csvdisplay precedes the data values by a header line listing the *k* indices and *n* expressions by name. If $csvdisplay_header is 0, this header line is omitted.

A.17 Reading data

The read command provides a way of reading unformatted data into AMPL parameters and other components, with syntax similar to the print command:

read [*indexing* :] *arglist* *redirection*$_{opt}$;

As with print, the optional *indexing* causes the read command to be executed separately for each member of the specified indexing set.

The read command treats its input as an unformatted series of data values, separated by white space. The *arglist* is a comma-separated list of arguments that specifies a series of components to which these values are assigned. As with print, the *arglist* is a comma-separated list of *args*, which may be any of

arg:
component-ref
indexing-expr component-ref
indexing-expr (*arglist*)

The *component-ref* must be a reference to a possibly suffixed parameter, variable, or constraint, or a suffixed problem name; it is meaningless to read a value into a set member or any more general expression. All indexing must be explicit, for example `read {j in DEST} demand[j]` rather than `read demand`. Values are assigned to *args* in the order that they are read, so later arguments can use values read earlier in the same `read` command.

If no *redirection* is specified, values are read from the current input stream. Thus if the `read` command was issued at an AMPL prompt, one types the values at subsequent prompts until all of the *arglist* entries have been assigned values. The prompt changes from `ampl?` back to `ampl:` when all the needed input has been read. If instead `read` is inside a script file that is read with `include` or `commands`, then the input is read from the same file, beginning directly after the `;` that ends the `read` command.

Most often the input to `read` lies in a separate file, specified by the optional *redirection*; its form is <*filename*, where *filename* is a string or parenthesized string expression that identifies a file. Multiple `read`'s can access the same file, in which case each `read` starts reading the file where the previous one left off. To force reading to start at the beginning again, `close` *filename* before re-reading.

If a script is to contain a `read` command that reads values typed interactively, the source of the values must be redirected to the standard input; specifying a `-` (minus sign) as the *filename* does so. This is most often used to read interactive responses from a user.

A.18 Modeling commands

A.18.1 The `solve` command

The `solve` command has the form

`solve` *redirection*$_{opt}$ `;`

It causes AMPL to write the current translated problem to temporary files in directory `$TMPDIR` (unless the current optimization problem has not changed since a previous `write` command), to invoke a solver, and to attempt to read optimal primal and dual variables from the solver. If this succeeds, the optimal variable values become available for printing and other uses. The optional *redirection* is for the solver's standard output.

The current value of the `solver` option determines the solver that is invoked. Appending `'_oopt'` to `$solver` gives the name of an option which, if defined with a nonempty string, determines (by the first letter of the string) the style of temporary problem files written; otherwise, AMPL uses its generic binary output format (style `b`). For example, if `$solver` is `supersol`, then `$supersol_oopt`, if nonempty, determines the output style. The command-line option `'-o?'` (A.23) shows a summary of the currently supported output styles.

AMPL passes two command-line arguments to the solver: the *stub* of the temporary files, and the literal string `-AMPL`. AMPL expects the solver to write dual and primal variable values to file *stub*`.sol`, preceded by commentary that, if appropriate, reports the objective value to

$objective_precision significant digits. In reading the solution, AMPL rounds the primal variables to $solution_round places past the decimal point if $solution_round is an integer, or to $solution_precision significant figures if $solution_precision is a positive integer; the defaults for these options imply that no rounding is performed.

A variable always has a current value. A variable declaration or data section can specify the initial value, which is otherwise 0. The option reset_initial_guesses controls the initial guess conveyed to the solver. If option reset_initial_guesses has its default value of 0, then the current variable values are conveyed as the initial guess. Setting option reset_initial_guesses to 1 causes the original initial values to be sent. Thus $reset_initial_guesses affects the starting guess for a second solve command, as well as for an initial solve command that follows a solution command (described below).

A constraint always has an associated current dual variable value (Lagrange multiplier). The initial dual value is 0 unless otherwise given in a data section or specified in the constraint's declaration by a := *initial_dual* or a default *initial_dual* phrase. Whether a dual initial guess is conveyed to solvers is governed by the option dual_initial_guesses. Its default value of 1 causes AMPL to pass the current dual variables (if $reset_initial_guesses is 0) or the original initial dual variables to the solver; if $dual_initial_guesses is set to 0, AMPL will omit initial values for the dual variables.

AMPL's presolve phase computes two sets of bounds on variables. The first set reflects any sharpening of the declared bounds implied by eliminated constraints. The other set incorporates sharpenings of the first set that presolve deduces from constraints it cannot eliminate from the problem. The problem has the same solutions with either set of bounds, but solvers that use active-set strategies (such as the simplex method) may have more trouble with degeneracies with the sharpened bounds. Solvers often run faster with the first set, but sometimes run faster with the second. By default, AMPL passes the first set of bounds, but if option var_bounds is 2, AMPL passes the second set. The .lb and .ub suffixes for variables always reflect the current setting of $var_bounds; .lb1 and .ub1 are for the first set, .lb2 and .ub2 for the second set.

If the output style is m, AMPL writes a conventional MPS file, and the value of option integer_markers determines whether integer variables are indicated in the MPS file by 'INTORG' and 'INTEND' 'MARKER' lines. By default, $integer_markers is 1, causing these lines to be written; specifying

```
option integer_markers 0;
```

causes AMPL to omit the 'MARKER' lines.

The option relax_integrality causes integer and binary attributes of variables to be ignored in solve and write commands. It is also possible to control this by setting the .relax suffix of a variable (A.11).

By default, values of suffixes of type IN or INOUT (A.11.1) are sent to the solver, and updated values for suffixes of type OUT or INOUT are obtained from the solver, but the sending and receiving of suffix values can be controlled by setting option send_suffixes suitably: if $send_suffixes is 1 or 3, suffix values are sent to the solver; and if $send_suffixes is 2 or 3, then updated suffix values are requested from the solver.

Whether .sstatus values (A.11.2) are sent to the solver is determined by options send_suffixes and send_statuses; setting $send_statuses to 0 causes .sstatus values not to be sent when $send_suffixes permits sending other suffixes.

Most solvers supply a value for AMPL's built-in symbolic parameter solve_message. AMPL prints the updated solve_message by default, but setting option solver_msg to 0

suppresses this printing. Most solvers also supply a numeric return code `solve_result_num`, which has a corresponding symbolic value `solve_result` that is derived from `solve_result_num` and `$solve_result_table` analogously to symbolic suffix values (A.11.1).

By default AMPL permutes variables and constraints so solvers see the nonlinear ones first. Some solvers require this, but with other solvers, occasionally it is useful to suppress these permutations by setting option `nl_permute` suitably. It is the sum of

1	to permute constraints
2	to permute variables
4	to permute objectives

and its default value is 3.

When complementarity constraints are present, the system of constraints is considered *square* if the number of ''inequality complements inequality'' constraints plus the number of equations equals the number of variables. Some complementarity solvers require square systems, so by default AMPL warns about nonsquare systems. This can be changed by adjusting option `compl_warn`, which is the sum of

1	warn about nonsquare complementarity systems
2	warn and regard nonsquare complementarity systems as infeasible
4	disregard explicit matchings of variables to equations

A.18.2 The `solution` command

The `solution` command has the form

`solution` *filename* ;

This causes AMPL to read primal and dual variable values from *filename*, as though written by a solver during execution of a `solve` command.

A.18.3 The `write` command

The `write` command has the form

`write` *outopt-value*$_{opt}$;

in which the optional *outopt-value* must adhere to the quoting rules for a *filename*. If *outopt-value* is present, `write` sets `$outopt` to *outopt-value*. Whether or not *outopt-value* is present, `write` then writes the translated problem as `$outopt` dictates: the first letter of `$outopt` gives the output style (A.18.1), and the rest is used as a ''stub'' to form the names of the files that are created. For example, if `$outopt` is `"b/tmp/diet"`, the `write` command will create file `/tmp/diet.nl`, and if `$auxfiles` so dictates (A.18.4), auxiliary files `/tmp/diet.row`, `/tmp/diet.col`, and so forth. The `solve` command's rules for initial guesses, bounds, suffixes, etc., apply.

A.18.4 Auxiliary files

The `solve` and `write` commands may cause AMPL to write auxiliary files. For the `solve` command, appending `_auxfiles` to `$solver` gives the name of an option that governs the auxiliary files written; for the `write` command, `$auxfiles` plays this role. The auxiliary files

Key	File	Description
a	*stub*.adj	constant added to objective values
c	*stub*.col	AMPL names of variables the solver sees
e	*stub*.env	environment (written by a `solve` command)
f	*stub*.fix	variables eliminated from the problem because their values are known
p	*stub*.spc	MINOS ''specs'' file for output style `m`
r	*stub*.row	AMPL names of constraints and objectives the solver sees
s	*stub*.slc	constraints eliminated from the problem
u	*stub*.unv	unused variables

Table A-12: Auxiliary files.

shown in Table A-12 are written only if the governing option's value contains the indicated key letter. If a key letter is capitalized, the corresponding auxiliary file is written only if the problem is nonlinear.

A.18.5 Changing a model: `delete`, `purge`, `redeclare`

The command

```
delete namelist ;
```

deletes each *name* in *namelist*, restoring any previous meaning *name* had, provided no other entities depend on *name*, i.e., if `xref` *name* (A.19.2) reports no dependents.

The command

```
purge namelist ;
```

deletes each *name* and all its direct and indirect dependents.

The statement

```
redeclare entity-declaration ;
```

replaces any existing declaration of the specified entity with the given one, provided either that the entity has no dependents, or that the new declaration does not change the character of the entity (its kind, such as `set` or `param`, and its number of subscripts). A `redeclare` can be applied to statements beginning with any of the following:

```
arc      function   minimize   param     set          suffix   var
check    maximize   node       problem   subject to   table
```

Redeclarations that would cause circular dependencies are rejected.

The command

```
delete check n;
```

deletes the *n*th check, while

```
redeclare check n indexing_opt : ... ;
```

redeclares the *n*th check.

A.18.6 The `drop`, `restore` and `objective` commands

These commands have the forms

```
drop indexing_opt constr-or-obj-name redirection_opt ;
restore indexing_opt constr-or-obj-name redirection_opt ;
objective objective-name redirection_opt ;
```

where *constr-or-obj-name* is the possibly subscripted name of a constraint or objective. The `drop` command instructs AMPL not to transmit the indicated entity (in `write` and `solve` commands); the `restore` command cancels the effect of a corresponding `drop` command. If *constr-or-obj-name* is not subscripted but is the name of an indexed collection of constraints or objectives, `drop` and `restore` affect all members of the collection.

The `objective` command arranges that only the named objective is passed to the solver. Issuing an `objective` command is equivalent to dropping all objectives, then restoring the named objective.

A.18.7 The `fix` and `unfix` commands

These commands have the forms

```
fix indexing_opt varname [ := expr ] redirection_opt ;
unfix indexing_opt varname [ := expr ] redirection_opt ;
```

where *varname* is the possibly subscripted name of a variable. The `fix` command instructs AMPL to treat the indicated variable (in `write` and `solve` commands) as though fixed at its current value, i.e., as a constant; the `unfix` command cancels the effect of a corresponding `fix` command. If *varname* is not subscripted but is the name of an indexed collection of variables, `fix` and `unfix` affect all members of the collection.

An optional `:=` *expr* may appear before the terminating semicolon, in which case the expression is assigned to the variable being fixed or unfixed, as though assigned by `let` (A.18.9).

A.18.8 Named problems and environments

The `problem` declaration/command has three functions: declaring a new problem, making a previously declared problem current, and printing the name of the current problem (in the form of a `problem` command establishing the current problem).

```
problem name indexing_opt environ_opt suffix-initializations_opt : itemlist ;
```

declares a new problem and specifies the variables, constraints, and objectives that are in it. Other variables appearing in the specified constraints and objectives are fixed (but can be unfixed by the `unfix` command). The new problem becomes the current problem. Initially the current problem is `Initial`. The *itemlist* in a problem declaration is a comma-separated list of possibly subscripted names of variables, constraints, and objectives, each optionally preceded by indexing. *Suffix-initializations* are analogous to those in constraint declarations, except that they appear before the colon.

The command

```
problem name ;
```

makes *name* (a previously declared problem) current, and

```
problem redirection_opt ;
```

prints the current problem name. Drop/restore and fix/unfix commands apply only to the current problem. Variable values, like parameters, are global; just the fixed/unfixed status of a variable depends on the problem. Similarly, the drop/restore status of a constraint depends on the problem (as do reduced costs). The current problem does not restrict the `let` command (A.18.9).

When a problem is declared, it can optionally specify an environment associated with the problem: the *environ* phrase has the form

```
environ envname
```

to specify that the problem's initial environment is *envname*, which must bear a subscript if the environment is indexed. Otherwise a new unindexed environment with the same name as the problem is created, and it inherits the then current environment (set of option values).

In `option` commands, unadorned (conventional) option names refer to options in the current environment, and the notation *envname*`.`*opname* refers to `$`*opname* in environment *envname*. The declaration

```
environ envname indexing_opt ;
```

declares environment *envname* (or indexed set of environments, if *indexing* is present). If there is no *indexing*, *envname* becomes the current environment for the current problem.

For previously declared environments, the command

```
environ envname ;
```

makes the indicated environment current, and the command

```
environ indexing_opt envname := envname_1 ;
```

copies environment $envname_1$ to *envname*, where *envname* and $envname_1$ must be subscripted if declared with *indexing*s. The initial environment is called `Initial`.

A.18.9 Modifying data: `reset`, `update`, `let`

The `reset` command has several forms.

```
reset ;
```

causes AMPL to forget all model declarations and data and to close all files opened by redirection, while retaining the current option settings.

```
reset options ;
```

causes AMPL to restore all options to their initial state. It ignores the current `$OPTIONS_IN` and `$OPTIONS_INOUT`; the files they name can be included manually, if desired.

```
reset data ;
```

causes AMPL to forget all assignments read in data mode and allows reading data for a different problem instance.

```
reset data name-list ;
```

causes AMPL to forget any data read for the entities in *name-list*; commas between names are optional.

A `reset data` command forces recomputation of all = expressions, and `reset data p`, even when `p` is declared with a = expression, forces recomputation of random functions in the = expression (and of any user-defined functions in the expression).

Problems (including the current one) are adjusted when their indexing expressions change, except that previous explicit drop/restore and fix/unfix commands remain in effect. The `reset problem` command cancels this treatment of previous explicit drop, restore, fix, and unfix commands, and brings the problem to its declared drop/fix state. This command has the forms

```
reset problem ;                          applies to the current problem
reset problem probname [subscript]opt ;
```

If the latter form mentions the current problem, it has the same effect as the first form. `reset problem` does not affect the problem's environment.

```
update data ;
```

permits all data to be updated once in subsequent data sections: current values may be overwritten, but no values are discarded.

```
update data name-list ;
```

grants update permission only to the entities in *name-list*.

The `let` command

```
let indexingopt name := expr ;
```

changes the value of the possibly indexed set, parameter or variable *name* to the value of the expression. If *name* is a set or parameter, its declaration must not specify a = phrase.

The command

```
let indexingopt name.suffix := expr ;
```

assigns the corresponding suffix value, if permitted. Some suffix values are derived and cannot be assigned; attempting to do so causes an error message.

A.19 Examining models

A.19.1 The `show` command

The command

```
show namelistopt redirectionopt ;
```

lists all model entities if *namelist* is not present. It shows each *name*'s declaration if it has one, or else lists model entities of the kind indicated by the first letters of each *name*:

```
ch... ==> checks              c...  ==> constraints
e...  ==> environments        f...  ==> functions
o...  ==> objectives
pr... ==> problems            p...  ==> parameters
su... ==> suffixes            s...  ==> sets
t...  ==> tables              v...  ==> variables
```

A.19.2 The `xref` command

The command `xref` shows entities that depend directly or indirectly on specified entities:

`xref` *itemlist redirection*$_{opt}$ `;`

A.19.3 The `expand` command

The `expand` command prints generated constraints and objectives:

`expand` *indexing*$_{opt}$ *itemlist redirection*$_{opt}$ `;`

The *itemlist* can assume the same forms allowed in problem declarations. If it is empty, all non-dropped constraints and objectives are expanded. The variant

`solexpand` *indexing*$_{opt}$ *itemlist redirection*$_{opt}$ `;`

shows how constraints and objectives appear to the solver. It omits constraints and variables eliminated by presolve unless they are explicitly specified in the *itemlist*.

Both the `expand` and `solexpand` commands permit variables to appear in the *itemlist*; for each, the commands show the linear coefficients of the variable in the relevant (non-dropped and, for `solexpand`, not eliminated by presolve) constraints and objectives, and indicates `"+ nonlinear"` when the variable also participates nonlinearly in a constraint or objective.

The options `expand_precision` and `expand_round` control printing of numbers by `expand`. By default they are printed to 6 significant figures.

A.19.4 Generic names

AMPL provides a number of generic names that can be used to access model entities without using model-specific names. Some of these names are described in Table A-13; the complete current list is on the AMPL web site.

These synonyms and sets can be used in `display` and other commands. They present the modeler's view (before presolve). Similar automatically updated entities with _ changed to `_s` (i.e., `_snvars`, `_svarnames`, `_svar`, etc.) give the solver's view, i.e., the view after presolve. There are exceptions, however, due to the way complementarity constraints are handled (A.9.1): none of `_cvar`, `_scconname`, or `_snccons` exists.

A.19.5 The `check` command

The command

```
check;
```

causes all `check` statements to be evaluated.

A.20 Scripts and control flow statements

AMPL provides statements similar to control flow statements in conventional programming languages, which make it possible to write a program of statements to be executed automatically.

`_nvars`	number of variables in current model
`_ncons`	number of constraints in current model
`_nobjs`	number of objectives in current model
`_varname{1.._nvars}`	names of variables in current model
`_conname{1.._ncons}`	names of constraints in current model
`_objname{1.._nobjs}`	names of objectives in current model
`_var{1.._nvars}`	synonyms for variables in current model
`_con{1.._ncons}`	synonyms for constraints in current model
`_obj{1.._nobjs}`	synonyms for objectives in current model
`_PARS`	set of all declared parameter names
`_SETS`	set of all declared set names
`_VARS`	set of all declared variable names
`_CONS`	set of all declared constraint names
`_OBJS`	set of all declared objective names
`_PROBS`	set of all declared problem names
`_ENVS`	set of all declared environment names
`_FUNCS`	set of all declared user-defined functions
`_nccons`	number of complementarity constraints before presolve
`_cconname{1.._nccons}`	names of complementarity constraints
`_scvar{1.._sncons}`	if `_scvar[`i`]` > 0, `_svar[scvar[`i`]]` complements `_scon[`i`]`
`_snbvars`	number of binary (0,1) variables
`_snccons`	number of complementarity constraints after presolve
`_snivars`	number of general integer variables (excluding binaries)
`_snlcc`	number of linear complementarity constraints
`_snlnc`	number of linear network constraints
`_snnlcc`	number of nonlinear compl. constrs.: `_snccons=_snlcc+_snnlcc`
`_snnlcons`	number of nonlinear constraints
`_snnlnc`	number of nonlinear network constraints
`_snnlobjs`	number of nonlinear objectives
`_snnlv`	number of nonlinear variables
`_snzcons`	number of constraint Jacobian matrix nonzeros
`_snzobjs`	number of objective gradient nonzeros

Table A-13: Generic synonyms and sets.

A.20.1 The `for`, `repeat` and `if-then-else` statements

Several commands permit conditional execution of and looping over lists of AMPL commands:

```
if lexpr then cmd
if lexpr then cmd else cmd
for loopname_opt indexing cmd
repeat loopname_opt opt-while { cmds } opt-while ;
break loopname_opt ;
continue loopname_opt ;
```

In these statements, *cmd* is either a single, possibly empty, command that ends with a semicolon or a sequence of zero or more commands enclosed in braces. *lexpr* is a logical expression. *loopname* is an optional loop name (which must be unbound before the syntactic start of the loop), which goes out of scope after syntactic end of the loop. If present, an *opt-while* condition has one of the forms

```
while lexpr
until lexpr
```

If *loopname* is specified, `break` and `continue` apply to the named enclosing loop; otherwise they apply to the immediately enclosing loop. A `break` terminates the loop, and `continue` causes its next iteration to begin (if permitted by the optional initial and final *opt-while* clauses of a `repeat` loop, or by the *indexing* of a `for` loop). Dummy indexes from *indexing* may appear in *cmd* in a `for` loop. The entire index set of a `for` loop is instantiated before starting execution of the loop, so the set of dummy indices for which the loop body is executed will be unaffected by assignments in the loop body.

Variants of `break`,

```
break commands ;
break all ;
```

terminate, respectively, the current `commands` command or all `commands` commands, if any, and otherwise act as a `quit` command.

Loops and if-then-else structures are treasured up until syntactically complete. Because `else` clauses are optional, AMPL must look ahead one token to check for their presence. At the outermost level, one must thus issue a null command (just a semicolon) or some other command or declaration to execute an outermost ‘‘`else`-less’’ `if` statement. (In this regard, end-of-file implies an implicit null statement.)

A semicolon is taken to appear at the end of command files that end with a compound command with optional final parts missing:

```
repeat ...  { ... }     # no final condition or semicolon
if ... then { ... }     # no else clause
```

AMPL has three pairs of prompts whose text can be changed through option settings. The default settings are:

```
option cmdprompt1 '%s ampl: ';
option cmdprompt2 '%s ampl? ';
option dataprompt1 'ampl data: ';
option dataprompt2 'ampl data? ';
option prompt1 'ampl: ';
option prompt2 'ampl? ';
```

`prompt1` appears when a new statement is expected, and `prompt2` when the previous input line is not yet a complete command (for example, if the semicolon at the end is missing).

In data mode, the values of `dataprompt1` and `dataprompt2` are used instead. When a new line is begun in the middle of an `if`, `for` or `repeat` statement, the values of `cmdprompt1` and `cmdprompt2` are used, with `%s` replaced by the appropriate command name; for example:

```
ampl: for {t in time} {
for{...} { ? ampl: if t <= 6
for{...} { ? ampl?   then let cmin[t] := 3;
if ... then {...} ? ampl: else let cmin[t] := 4;
for{...} { ? ampl: };
ampl:
```

A.20.2 Stepping through commands

It is possible to step through commands in an AMPL script one command at a time. Single-step mode is enabled by

```
option single_step n ;
```

where *n* is a positive integer; it specifies that if the insert level is at most *n*, AMPL should behave as though `commands -;` were inserted before each command: it should read commands from the standard input until `end` or other end of file signal (control-D on Unix, control-Z on Windows). Some special commands may appear in this mode:

`step` n_{opt}	execute the next command, or *n* commands
`skip` n_{opt}	skip the next command, or *n* commands
`next` n_{opt}	if the next command is an if-then-else or looping command, execute the entire compound command, or *n* commands, before stopping again (unless the compound command itself specifies `commands -;`)
`cont`	execute until the end of all currently nested compound commands at the current insert level

A.21 Computational environment

AMPL runs in an operating system environment, most often as a standalone program, but sometimes behind the scenes in a graphical user interface or a larger system. Its behavior is influenced by values from the external environment, and it can set values that become part of that environment. The parameter `_pid` gives the process ID of the AMPL process (a number unique among processes running on the system).

A.21.1 The `shell` command

The `shell` command provides a temporary escape to the operating system, if such is permitted, to run commands.

```
shell 'command-line' redirection_opt ;
shell redirection_opt ;
```

The first version runs *command-line*, which is contained in a literal string. In the second version, AMPL invokes an operating-system shell, and control returns to AMPL when that shell terminates. Before invoking the shell, AMPL writes a list of current options and their values to the file (if any) named by option `shell_env_file`. The name of the shell program is determined by option `SHELL`.

A.21.2 The `cd` command

The `cd` command reports or changes AMPL's working directory.

```
cd ;
cd new-directory ;
```

The parameter `_cd` is set to this value.

`_ampl_elapsed_time`	elapsed seconds since the start of the AMPL process
`_ampl_system_time`	system CPU seconds used by the AMPL process itself
`_ampl_user_time`	user CPU seconds used by the AMPL process itself
`_ampl_time`	`_ampl_system_time + _ampl_user_time`
`_shell_elapsed_time`	elapsed seconds for most recent shell command
`_shell_system_time`	system CPU seconds used by most recent shell command
`_shell_user_time`	user CPU seconds used by most recent shell command
`_shell_time`	`_shell_system_time + _shell_user_time`
`_solve_elapsed_time`	elapsed seconds for most recent solve command
`_solve_system_time`	system CPU seconds used by most recent solve command
`_solve_user_time`	user CPU seconds used by most recent solve command
`_solve_time`	`_solve_system_time + _solve_user_time`
`_total_shell_elapsed_time`	elapsed seconds used by all shell commands
`_total_shell_system_time`	system CPU seconds used by all shell commands
`_total_shell_user_time`	user CPU seconds used by all shell commands
`_total_shell_time`	`_total_shell_system_time+_total_shell_user_time`
`_total_solve_elapsed_time`	elapsed seconds used by all solve commands
`_total_solve_system_time`	system CPU seconds used by all solve commands
`_total_solve_user_time`	user CPU seconds used by all solve commands
`_total_solve_time`	`_total_solve_system_time+_total_solve_user_time`

Table A-14: Built-in timing parameters.

A.21.3 The `quit`, `exit` and `end` commands

The `quit` command causes AMPL to stop without writing any files implied by `$outopt`, and the `end` command causes AMPL to behave as though it has reached the end of the current input file, without reverting to model mode. At the top level of command interpretation, either command terminates an AMPL session. The command `exit` is a synonym for `quit`, but it can return a status to the surrounding environment:

`exit` *expression*$_{opt}$ `;`

A.21.4 Built-in timing parameters

AMPL has built-in parameters that record various CPU and elapsed times, as shown in Table A-14. Most current operating systems keep separate track of two kinds of CPU time: *system* time spent by the operating time on behalf of a process, e.g., for reading and writing files, and *user* time spent by the process itself outside of the operating system. Usually the system time is much smaller than the user time; when not, finding out why not sometimes suggests ways to improve performance. Because seeing separate system and user times can be helpful when performance seems poor, AMPL provides built-in parameters for both sorts of times, as well as for their sums. AMPL runs both solvers and shell commands as separate processes, so it provides separate parameters to record the times taken by each sort of process, as well as for the AMPL process itself.

A.21.5 Logging

If option `log_file` is a nonempty string, it is taken as the name of a file to which AMPL copies everything it reads from the standard input. If option `log_model` is 1, then commands and

declarations read from other files are also copied to the log file, and if `log_data` is 1, then data sections read from other files are copied to the log file as well.

A.22 Imported functions

Sometimes it is convenient to express models with the help of functions that are not built into AMPL. AMPL has facilities for importing functions and optionally checking the consistency of their argument lists. **Note:** The practical details of using imported functions are highly system-dependent. This section is concerned only with syntax; specific information will be found in system-specific documentation, e.g., on the AMPL web site.

An imported function may need to be evaluated to translate the problem; for instance, if it plays a role in determining the contents of a set, AMPL must be able to evaluate the function. In this case the function must be linked, perhaps dynamically, with AMPL. On the other hand, if an imported function's only role is in computing the value of a constraint or objective, AMPL never needs to evaluate the function and can simply pass references to it on to a (nonlinear) solver.

Imported functions must be declared in a `function` declaration before they are referenced. This statement has the form

`function` *name* *alias*$_{opt}$ (*domain-spec*)$_{opt}$ *type*$_{opt}$ [`pipe` *litseq*$_{opt}$ [`format` *fmt*]] ;

in which *name* is the name of the function, and *domain-spec* amounts to a function prototype:

domain-spec:
    domain-list
    `...`
    nonempty-domain-list , `...`

A *domain-list* is a (possibly empty, comma-separated) list of set expressions, asterisks (*'s), direction words (`IN`, `OUT`, or `INOUT`), direction words followed by set expressions, and *iterated-domain-list*s:

iterated-domain-list:
    indexing (*nonempty domain-list*)

An *iterated-domain-list* is equivalent to one repetition of its *domain-list* for each member in the *indexing* set, and the domain of dummy variables appearing in the *indexing* extends over that *domain-list*. The direction words indicate which way information flows: into the function (`IN`), out of the function (`OUT`), or both, with `IN` the default. In a function invocation, `OUT` arguments are assigned values specified by the function at the end of the command invoking the function.

Omitting the optional (*domain-spec*) in the `function` declaration is the same as specifying `(...)`. The function must be invoked with at least or exactly as many arguments as there are sets in the *domain-spec* (after *iterated-domain-list*s have been expanded), depending on whether or not the *domain-spec* ends with `...`. AMPL checks that each argument corresponding to a set in the *domain-list* lies in that set. A * by itself in a *domain-list* signifies no domain checking for the corresponding argument.

A function whose return value is not of interest can be invoked with a `call` command:

`call` *funcname*`(` *arglist* `);`

Type can be `symbolic` or `random` or both; `symbolic` means the function returns a literal (string) value rather than a numeric value, and `random` indicates that the ''function'' may return

different values for the same arguments, i.e., AMPL should assume that each invocation of the function returns a different value.

The commands

```
load     libnames_opt ;
unload   libnames_opt ;
reload   libnames_opt ;
```

load, unload, or reload shared libraries (from which functions and table handlers are imported); *libnames* is a comma-separated list of library names. When at least one *libname* is mentioned in the `load` and `unload` commands, `$AMPLFUNC` is modified to reflect the full pathnames of the currently loaded libraries. The `reload` command first unloads its arguments, then loads them. This can change the order of loaded libraries and affect the visibility of imported functions: the first name wins. With no arguments, `load` loads all the libraries currently in `$AMPLFUNC`; `unload` unloads all currently loaded libraries, and `reload` reloads them (which is useful if some have been recompiled).

The keyword `pipe` indicates that this is a *pipe function*, which means AMPL should start a separate process for evaluating the function. Each time a function value is needed, AMPL writes a line of arguments to the function process, then reads a line containing the function value from the process. (Of course, this is only possible on systems that allow multiple processes.) A *litseq* is a sequence of one or more adjacent literals or parenthesized string expressions, which AMPL concatenates and passes to the operating system (i.e., to `$SHELL`) as the description of the process to be invoked. In the absence of a *litseq*, AMPL passes a single literal, whose value is the name of the function. If the optional `format` *fmt* is present, *fmt* must be a format, suitable for `printf`, that tells AMPL how to format each line it sends to the function process. If no *fmt* is specified, AMPL uses spaces to separate the arguments it passes to the pipe function.

For example:

```
ampl: function mean2 pipe "awk '{print ($1+$2)/2}'";
ampl: display mean2(1,2) + 1;
mean2(1, 2) + 1 = 2.5
```

The function `mean2` is expected (by default) to return numeric values; AMPL will complain if it returns a string that does not represent a number.

The following functions are `symbolic`, to illustrate formatting and the passing of arguments.

```
ampl: function f1 symbolic pipe "awk '" '{printf "x%s\n", $1}' "'";
ampl: function g1 symbolic pipe 'awk ''{printf "XX%s\n", $1}''';
ampl: function cat symbolic pipe format ">>%s<<\n";

ampl: display f1(2/3);
f1(2/3) = x0.6666666666666667

ampl: display g1('abc');
g1('abc') = XXabc

ampl: display cat('some words');
cat('some words') = ">>'some words'<<"
```

The declaration of `f1` specifies a *litseq* of 3 literals, while `g1` specifies one literal; `cat`, having an empty *litseq*, is treated as though its *litseq* were `'cat'`. The literals in each *litseq* are stripped of the quotes that enclose them, have one of each adjacent pair of these quotes removed, and have (*backslash*, *newline*) pairs changed to a single newline character; the results are concatenated to

produce the string passed to the operating system as the description of the process to be started. Thus for the four pipe functions above, the system sees the commands

```
awk '{print ($1+$2)/2}'
awk '{printf "x%s\n", $1}'
awk '{printf "XX%s\n", $1}'
cat
```

respectively. Function `cat` illustrates the optional `format` *fmt* phrase. If *fmt* results in a string that does not end in a newline, AMPL appends a newline character. If no *fmt* is given, each numeric argument is converted to the shortest decimal string that rounds to the argument value.

Caution: The line returned by a pipe function must be a complete line, i.e., must end with a *newline* character, and the pipe function process must flush its buffers to prevent deadlock. (Pipe functions do not work with most standard Unix programs, because they don't flush output at the end of each line.)

Imported functions may be invoked with conventional functional notation, as illustrated above. In addition, iterated arguments are allowed. More precisely, if `f` is an imported function, an invocation of `f` has the form `f(`*arglist*`)` in which *arglist* is as for the `print` and `printf` commands — a possibly empty, comma-separated list of expressions and *iterated-arglist*s:

```
ampl: function mean pipe 'awk ''{x = 0\
        for(i = 1; i <= NF; i++) x += $i\
        printf "%.17g\n", x/NF}''';
ampl: display mean({i in 1..100} i);
mean({i in 1 .. 100} (i)) = 50.5

ampl: display mean({i in 1..50}(i,i+50));
mean({i in 1 .. 50} (i, i + 50)) = 50.5

ampl: display mean({i in 0..90 by 10}({j in 1..10} i + j));
mean({i in 0 .. 90 by 10} ({j in 1 .. 10} (i + j))) = 50.5
```

The command

```
reset function name_opt ;
```

closes all pipe functions, causing them to be restarted if invoked again. If an function is named explicitly, only that function is closed.

A.23 AMPL invocation

AMPL is most often invoked as a separate command in some operating system environment. When AMPL begins execution, the declarations, commands, and data sections described above (A.14) can be entered interactively. Depending on the operating system where AMPL is run, the invocation may be accompanied by one or more *command-line arguments* that set various properties and options and specify files to be read. These can be examined by typing the command

```
ampl '-?'
```

The initialization of some options may be determined by command-line arguments. The `'-?'` argument produces a listing of these options and their command-line equivalents.

Sometimes it is convenient to have option settings remembered across AMPL sessions. Under operating systems from which AMPL can inherit environment variables as described above, the

options OPTIONS_IN, OPTIONS_INOUT, and OPTIONS_OUT provide one way to do this. If $OPTIONS_IN is nonempty in the inherited environment, it names a file (meant to contain option commands) that AMPL reads before processing command-line arguments or entering its command environment. OPTIONS_INOUT is similar to OPTIONS_IN; AMPL reads file $OPTIONS_INOUT (if nonempty) after $OPTIONS_IN. At the end of execution, if $OPTIONS_INOUT is nonempty, AMPL writes the current option settings to file $OPTIONS_INOUT. If nonempty, $OPTIONS_OUT is treated like $OPTIONS_INOUT at the end of execution.

The command-line argument -v prints the version of the AMPL command being used; this is also available as option version.

The command-line option -R (recognized only as the first command-line option and not mentioned in the -? listing of options) puts AMPL into a restricted ''server mode,'' in which it declines to execute cd and shell commands, forbids changes to options TMPDIR, ampl_include, and PATH (or the search path for the operating system being used), disallows pipe functions, and restricts names in option solver and file redirections to be alphanumeric (so they can only write to the current directory, which, on Unix systems at least, cannot be changed). By invoking AMPL from a shell script that suitably adjusts current directory and environment variables before it invokes ampl -R, one can control the directory in which AMPL operates and the initial environment that it sees.

On systems where imported function libraries can be used, the command-line option -i*libs* specifies libraries of imported functions (A.22) and table handlers (A.13) that AMPL should load initially. If -i*libs* does not appear, AMPL assumes -i$AMPLFUNC. Here *libs* is a string, perhaps extending over several lines, with the name of one library or directory per line. For a directory, AMPL looks for library amplfunc.dll in that directory. If *libs* is empty and amplfunc.dll appears in the current directory, AMPL loads amplfunc.dll initially. If library ampltabl.dll is installed in what the operating system considers to be a standard place, AMPL also tries to load this library, which can provide ''standard'' database handlers and functions.